Dirac 3

Cross-Platform Time and Pitch Manipulation Library for Polyphonic Audio

v3.5.4 [1202281700] • 27-02-2012

http://www.dspdimension.com

DiracLE
License Agreement

About this Manual

This file is provided as a reference and a manual for the Dirac library of functions. Please note that everything described herein is subject to change without notice.

We recommend you either read this manual sequentially if you're interested in all the bells and whistles of the various Dirac 3.5 APIs, or if you're ready to get your hands dirty you can start with chapter 4 which describes a simple command line application that uses the new DiracFx API to do time stretching and pitch shifting in the blink of an eye.

If you have any questions please don't hesitate to contact us via our contact form at http://www.dspdimension.com.

Available Dirac Versions and License Models

Desktop license

The Dirac desktop license includes libraries for Windows 32/64bit, Linux 32/64bit and MacOS X 32/64bit PPC+Intel.

There are 3 different incarnations of the desktop version of Dirac available at this time.

DiracLE is the free version of the library for mobile and desktop that provides the exact same audio quality as the STUDIO and PRO versions but has some minor usability restrictions. For example, DiracLE can only process one audio channel per Dirac instance. This doesn't mean that you cannot process stereo files with the LE version – you simply need to create one separate DiracLE object per channel.

Dirac STUDIO adds phase locked stereo processing and the PRO version allows processing an arbitrary number of channels (5.1 surround and more) in a phase locked, mono-compatible manner. PRO also includes the full-featured mobile license of Dirac.

	DiracLE	Dirac Studio	Dirac Pro
Supported Sample Rates	44.1kHz, 48kHz	8 kHz – 96kHz	unlimited
Dynamic (time varying) time stretch/pitch shift	NO	NO	YES
Max. supported no. of Channels	1	2	unlimited
Transcribe Mode	YES	YES	YES
DiracRetune API	NO	NO	YES
DiracFx API	YES	YES	YES
Includes License for Mobile Devices	YES	NO	YES
64bit compatible	NO	YES	YES

Mobile license

The Dirac PRO mobile license only contains the full set of features of the PRO version for ARM compliant processors, ie the iPhone/iPad ARM 6/7 version for iOS4 and higher. The mobile license is also available separately.

Source Code

We offer source code licensing in some special cases if object code cannot be used. Please inquire to see if this is the case with your project.

Use of DiracLE in a Commercial Product

For detailed information please see our Licensing Agreement at the beginning of this PDF file. In short, you may freely use DiracLE in a commercial application provided that you mention us as the author of the Dirac technology. The phrase we recommend is as follows:

```
“DIRAC Time Stretch / Pitch Shift technology licensed from The DSP
Dimension, http://www.dspdimension.com. DIRAC is (c) Stephan M. Bernsee”
```

Even though this is not stated in the Agreement you may abbreviate this to match the design and available space in your app, and the link to our web site is optional. However, please make sure you indicate that this text ONLY refers to the DiracLE technology and not your entire app, as your customers might otherwise be misled into believing that we are the original author of your app and they might contact us for help with your product.

Realtime Performance on Handheld Devices

We are introducing a new mode in Dirac v3.5 that allows polyphonic, transient preserving time stretching and pitch shifting at a slightly reduced quality but at an incredible speed. Speed measurements indicate that the new DiracFx API uses an average of 20% of the CPU speed on a 3GS – for a stereo instance. We recommend that you use the DiracFx API whenever speed and low power consumption is of utmost importance. For more information please see the section about the new DiracFx API.

Dirac itself has also been highly optimized to work very efficiently on a wide variety of hardware. If you want to use the fully featured Dirac core API we recommend you use the following settings for creating a Dirac instance on devices where CPU time and battery life is important, such as on net books and handheld devices such as tablets and cell phones:

```
mDirac = DiracCreate(kDiracLambdaPreview, kDiracQualityPreview, ...);
mDirac = DiracCreate(kDiracLambdaTranscribe, kDiracQualityPreview, ...);
```

Higher quality modes may be possible depending on processor speed. Also, if you are processing mostly speech or any other kind of vocal utterances, you could use the specialized voice mode:

```
mDirac = DiracCreate(kDiracLambda1, kDiracQualityPreview, ...);
```

Also due to the CPU intensive nature of all time stretching processes we recommend at least an iPhone 3GS or better running iOS 4 or later for smooth operation. Less powerful devices may be able to run Dirac in realtime if the sample rate and/or number of channels are reduced, but we don’t guarantee realtime performance on any iOS devices released prior to the 3GS.

What's New in Version 3.5?

Dirac v3.5 offers many enhancements over previous versions. Following is a list of changes and improvements as well as a description of API changes in comparison to version 3. For a comparative list of the version 2 -> 3 API changes please see previous versions of this document that is available with pre-3.5 versions of Dirac.

Please note that as of version 3.5 we no longer support iOS versions prior to iOS 4. Should you absolutely require an iOS 3.2 build please contact us via email and we'd be happy to help.

More Fine Grained Control

We have added a new callback option to Dirac v3.5 that allows a more fine grained control over Dirac's settings than was possible with previous versions. Our new `DiracSetProcessingBeganCallback()` installs a callback that gets invoked immediately before the Dirac core engine applies its DSP processing to the signal. You can use this callback to get/set speed, pitch and formant factors during processing (for instance, when evaluating a curve that controls speed, pitch or other properties depending on the location in file). See chapter 3 for more information on this call.

DiracFx API

A new easy-to-use API has been added to the Dirac library that allows transient preserving time stretching and pitch shifting of speech and polyphonic music at slightly reduced quality but at an incredible speed. Our measurements indicate that this mode utilizes only about 20% CPU on average for a stereo instance on the 3GS. This new mode was made possible by a novel technology called "eodymium" that the DSP Dimension has developed in 2010, which allows extensive time-frequency modifications while using no trigonometric functions at all.

DiracRetune

We have revisited the code behind our popular DiracRetune engine. DiracRetune is now easier to set up, less sensitive to noise and entirely sample rate independent. The changes we've made require adjusting the calls that you're using in your application. Some features of DiracRetune have been discontinued or deprecated. Please see the Dirac.h header file for the updated function prototypes.

Also, we have discontinued access to DiracRetune through the Dirac core API (kDiracLambdaRetune selector). This was inconsistent with our API, inefficient, and according to the feedback we got from our clients noone was actually using it that way. We no longer support effect mode either, as it never really worked reliably the way we intended.

It has been a popular request that the tuning table feature should behave like the key enable/disable feature that is popular with many products on the market that do pitch correction. Usually this is done by allowing and disallowing keys within an octave. We are therefore deprecating the call DiracRetuneSetTuningTable to reflect this change, and are introducing several new calls instead.

API Changes in v3.5:

Dirac Core API

Removed:
```
DiracProcessWithUserData (use DiracProcess)
DiracProcessInterleavedWithUserData (use DiracProcessInterleaved)
```

Added:
```
DiracSetProcessingBeganCallback
```

DiracRetune API

Changed:
```
DiracRetuneSetProperties
DiracRetuneCreate
```

Deprecated or removed:
```
DiracRetuneSetTuningTable (use DiracRetuneSetKeyStatus, DiracRetuneSetAllowedKeysMask,
                              DiracRetuneSetKeyList)
DiracRetuneSetTuningReferenceHz (use DiracRetuneCreate)
DiracRetuneSetPitchHz (use DiracRetuneSetKeyList)
```

Added:
```
DiracRetuneGetKeyStatus
DiracRetuneSetKeyStatus
DiracRetuneGetAllowedKeysMask
DiracRetuneSetAllowedKeysMask
DiracRetuneGetClosestKeyDetuneCent
DiracRetuneGetClosestKey
DiracRetunePrintInternalTuningTable
DiracRetuneSetKeyList
DiracRetuneLatencyFrames
```

DiracFx API

Added:
```
DiracFxCreate
DiracFxMaxOutputBufferFramesRequired
DiracFxOutputBufferFramesRequiredNextCall
DiracFxLatencyFrames
DiracFxDestroy
DiracFxProcessFloat
DiracFxProcessFloatInterleaved
DiracFxProcess
DiracFxProcessInterleaved
DiracFxReset
```

1

Chapter 1: Introduction

In today's audio processing applications, independent time and pitch manipulation has become an important feature for the creation, composition and manipulation of digital audio. Applications range from fitting a drum loop to a predefined tempo to creating backing vocals or dynamically tweaking the timing of a song to give it a different feel.

Ideally, such a time compression/expansion and pitch shifting process should be scalable to provide high quality or fast processing speed, it should preserve the relative timing of events within the audio stream and should work equally well with musically monophonic and polyphonic material within a stretch ratio of 0.5 to 2.0 (double and half speed).

In the past, there have been two major algorithm types in use. *Pitch Synchronized Overlap-Add* provided the best time localization (the least smearing of transients) but worked only with material that had a prominent fundamental frequency such as musically monophonic signals like voice. On the other hand, frequency domain methods such as the *improved Phase Vocoder*, while providing relatively high quality for musically polyphonic material, still suffer from transient smearing when used with voice or very percussive signals.

We're happy to introduce Dirac, the world's first high end time and pitch manipulation framework that addresses all these issues. Based on a novel time-frequency approach, Dirac can be seamlessly scaled between the high coherence of time domain methods and the excellent frequency localization properties of the phase vocoder. What's more, the basic DiracLE library comes completely free of charge so you can start using this intriguing functionality in all your audio related projects today – cross-platform, on MacOS X, Linux, Windows and the iPhone OS.

Dirac can be included in your project in the matter of minutes, is easy to set up and leaves you without worries about processing latency and long term accuracy. It has been extensively tested with a wealth of different audio signals and offers flexible control over its quality and localization properties. It is multi-channel- and multi-threading savvy and has been found to be extremely accurate, preserving both the relative timing of events and the stereo localization.

On top of this, Dirac supports both RAM- and disk-based applications and is very easy to include into any project since the actual processing is performed using only 4 calls to the Dirac core routines. You will find an example of how to include Dirac into your project along with a detailed description of its parameters below. The algorithm does not alter the timing structure of the audio material in any way thus providing an accurate time compression/expansion tool for applications where the relative timing of instruments or acoustic cues is critical (such as in drum loops).

Also, high quality pitch conversion for changing pitch without affecting sample length including full anti-aliasing has already been included and can be selected by simply setting the corresponding parameter to an appropriate value (s.b.).

1.1 Formant Preservation

One important feature of the Dirac core library is its capability to preserve the harmonic structure of a sound, making transposition artifacts such as the munchkinization effect a thing of the past. Dirac is capable of transposing a sound over a wide range of semitones without altering its timbre, making it an ideal tool for doubling vocal passages in a song, or altering the timbre or gender of a voice.

1.2 Dynamic (= time varying) Time Stretch/Pitch Shift[ü]

With Dirac PRO, you can dynamically alter the speed and pitch of your signal. As Dirac processes your file, simply change the parameters according to the user input. You can dynamically alter timing, pitch and formants of your audio signal, either together or completely independent from each other. That way you can create ritardandi or accelerandi, or glissandi when processing vocals.

1.3 How to Include Dirac Functionality into an Existing C++ Project

On the Windows platform, some versions of Dirac are supplied as Dynamic Link Library (DLL), which provides the easiest way of updating the Dirac component, making it independent of the actual host application that uses it. To use the Dirac functionality in your project add the Dirac.lib stub library to your project and make sure your compiler can find the associated Dirac.h header file. To use the Dirac routines, the Dirac.dll library must be located in the same directory as your application, or in any of the standard DLL directories so your host application can find the file.

On the Mac, Linux and the iPhone, Dirac is provided as a static library. To use the Dirac routines in your project, simply add the Dirac library file to your project, either by using the "Project -> Add -> Existing Files…" menu entry or by dragging the Library icon into the project workspace window. In case your compiler cannot find the Dirac.h header file make sure you add this to your project workspace as well.

Dirac utilizes the full potential of the PowerPC AltiVec (Velocity), Intel SSE and VFP/NEON engine where available. For this feature to work on the Mac, you will need to include the vecLib.framework and Accelerate.framework in your project). If you don't include these frameworks you might get link errors due to the missing vector functions, or the library might fall back to using scalar code.

Now open your C++ source file in which you plan to include the calls to the Dirac routines and include the "Dirac.h" header file at the beginning of your source text. Make sure the "Settings…" include directories are properly set so the IDE can find the files.

1.4 Definition of Terms

We will use the terms 'sample' and 'frame' in the context of this documentation. A *sample* is the basic unit of stored digital audio data. A *frame* is defined as consisting of 1 sample for each channel. One *stereo frame* therefore consists of two samples, one for the left and one for the right channel. A *selection* is a segment of frames selected by an user in the editor. We will avoid the term 'sample' in conjunction with sampled sounds, and adopt the term *data set* instead. A data set is a collection of frames uniformly sampled at an arbitrary sample rate, passed to the algorithm in the channel array.

1.5 About the Dirac Algorithm

Past research has shown time domain [pitch] synchronized overlap-add ([P]SOLA) algorithms for independent time and pitch manipulation of audio ("time stretching" and "pitch shifting") to be the method of choice for single-pitched sounds such as voice and musically monophonic instrument recordings due to the prominent periodicity at the fundamental period. On the other hand, frequency domain methods have recently evolved around the concept of the phase vocoder that have proven to be vastly superior for multi-pitched sounds and entire musical pieces.

Dirac is a cross-platform C/C++ object library that exploits the good localization of wavelets in both time and frequency to build an algorithm for time and pitch manipulation that uses an arbitrary time-frequency tiling depending on the underlying signal. Additionally, the time and frequency localization parameter of the basis can be user-defined, making the algorithm smoothly scalable to provide either the phase coherence properties of a time domain process or the good frequency resolution of the phase vocoder.

Dirac was developed by Stephan M. Bernsee, who has pioneered high quality time and pitch manipulation of polyphonic audio. In the 1990s, he developed the first algorithm on the market that allowed high quality time and pitch

[ü] Note that the commercial Dirac STUDIO or PRO retail version is required in order to use this feature.

manipulation of musically polyphonic signals which is still the standard in today's high-end commercial audio processing applications.

The basic DiracLE library comes as a free download off the DSPdimension web site and is currently available for Microsoft Visual C6+ and for Xcode/gcc on MacOS X and iPhone, and gcc on Linux.

1.6 About the Dirac Core API

As with all time stretching algorithms, Dirac operates on blocks (chunks) of data. In order to deal with the unequal number of frames at the input and the output that result from changing the length of your audio data Dirac utilizes a design pattern called an "unidirectional pull model" (it is unidirectional in that its communication is one way, ie. it does not rely on the sender to notify the receiver that new data is available – instead it asks the sender to supply data as needed).

This means that you call Dirac from your code to deliver chunks of processed sound whenever you need them in your app. In order to create these chunks Dirac in turn invokes a callback method whenever it requires new audio data from the underlying data source. This callback is code that you will write, so the actual sound source that delivers the audio data is totally unknown to Dirac, and you can deliver audio data in any way you want, either from memory, a file, a network connection or by creating sound programmatically. By the same token, Dirac knows nothing about file formats or codecs, all it really cares about is getting a stream of single precision float data from you to apply its processing to.

For more information and implementation please see our example projects and please see chapter 3 in this PDF for a detailed description of the API.

1.7 Can I License Source Code?

Our past experience shows that some companies don't want to license object code. Object code is a "black box" that can only be maintained by the original author, who might not always be available to provide support for an unlimited period of time, so this is considered a risk. On the other hand, object code like Dirac doesn't interface much (or at all) with system specific calls, which makes it very long-lived compared to the usual life span of an actual end user application.

Developers don't usually hand out source code, because they would give away control over their work and inventions. Once released, source code can be modified, re-sold, sub-licensed etc. so it's usually not in the author's interest to disclose it. For Dirac, you have the option of purchasing source code from me if you want to. Of course, this is a more expensive option than licensing object code, but the option is there.

Should you be interested in licensing source code please contact us via our web site at http://www.dspdimension.com. Please note that source code is only available commercially, We do not license source code for educational or research purposes – please use the freely available LE library in this case.

2

Chapter 2: Terms and Definitions

The Dirac routines are entirely independent of any operating system specific calls in that they do not contain any user interface code. However, Dirac needs to employ its own memory management to be free from any restrictions with regard to number of channels and sample rate. It uses the standard library malloc/calloc() and free() calls to do this. Note that no STL calls are being used so you don't have to worry about quirks of a particular version of STL. You need to make sure the project contains the standard libraries (stdlib and mathlib) providing the abovementioned functions.

On MacOS X and iOS you might need to add additional vector libraries to your project. Please see your compiler's manual for more information on using vecLib/Accelerate frameworks in your project.

2.1 Processing Delay (Latency) Considerations

The delay introduced between the start of the original data and the same block of data in the output file when processed in 1:1 (bypass) mode is usually referred to as the **processing delay**, or latency. With the Dirac ccore API we have created a processing framework that is essentially free from any delay – you don't have to worry about latency issues[1]

2.2 Quality, Speed and Localization Properties (Dirac Core API only)

2.2.1 Time/Frequency Localization and the 'lambda' Parameter

Dirac uses a novel algorithm that can be scaled to provide good time domain localization or good frequency localization, or both. High time localization means that Dirac produces results similar to the time domain pitch-synchronized overlap-add (PSOLA) methods, high frequency localization produces results that are closer to what you get from an improved phase vocoder.

This ability is controlled by one of two parameters called "*lambda*" which is set when you create the Dirac object. As a rule of thumb, a low Lambda value provides good time localization (good for voice and single instrument recordings), while a high lambda value is good for entire mixes. High lambda values take slightly more time to process but are not considerably slower.

The following lambda settings are available[2]:

Value	Description
`kDiracLambdaPreview`	This automatically selects the best time/frequency tradeoff for realtime/preview performance. It is the fastest setting but might not provide the best results in all cases.
`kDiracLambda1`	Selects full time localization. Good setting for single instruments and voice. Required setting for doing pitch correction.
`kDiracLambda2`	Time/frequency localization with emphasis on time localization. If a setting of `kDiracLambda1` produces echoes this might be a better

[1] Note that this applies to the Dirac core API only. Both the DiracRetune and the DiracFx APIs introduce an additional processing latency into the signal. Please see the relevant sections for more details.

[2] Note that creating a DiracRetune instance from within the Dirac core API has been discontinued in version 3.5. See the section on the DiracRetune API for more information.

	choice
kDiracLambda3	This sets the time/frequency localization halfway between time and frequency domains. It is the best setting for all general purpose signals and should be set as default for non-realtime (non-preview) processing.
kDiracLambda4	Higher frequency localization and less time localization. Might be a better choice for classical music than the lower-lambda settings
kDiracLambda5	Highest frequency localization. This might not be an ideal choice if you're dealing with signals that have very sharp attack transients but it might be useful for sensitive material such as classics
kDiracLambdaTranscribe	Special mode for very large stretch ratios (2x to 4x).

2.2.2 Processing Resolution and the 'quality' Parameter

The second parameter is used to set the *processing quality*. A low quality setting provides excellent performance at a slightly lower algorithm quality, while higher values render the results in more time but at a significantly higher resolution.

The following four quality settings are available:

Value	Description
kDiracQualityPreview	This quality mode offers preview quality, which is usually good enough for a preview to see the effects of the parameter settings or for realtime processing.
kDiracQualityGood	A better quality mode than kDiracQualityPreview. It is recommended as the default quality setting for non-realtime (non-preview) processing
kDiracQualitBetter	Very good quality mode but takes more CPU.
kDiracQualityBest	The highest quality mode. Note that this setting can be *very* slow.

2.3 Phase Locked Multi-Channel Processing vs. Multiple Channel Processing[ü]

The STUDIO version of Dirac supports stereo while Dirac PRO supports an infinite number of channels (memory permitting) that it can process in a phase-locked (synced) manner at the same time. All of these simultaneous channels are being processed using a phase-locked processing algorithm that ensures that the stereo (or surround/multi-channel) phase relationship is preserved.

It is important to understand how this works and what this means exactly.

In a stereo recording, important localization cues are provided to the listener through the relative timing of a sound source between the left and the right ear (channel). If a time stretching process changes the relative timing of the two channels by even a minimal amount, the stereo image will be perceived as "distorted". Also, mono compatibility will no longer be guaranteed, which means that if you mix down the two stereo channels to a mono channel (as is the case in some TV and radio equipment) you will end up with very audible artifacts perceived as phasing or even cancellations.

If you have the situation that the relative phase between channels matters, it is imperative to use the multi-channel processing mode of Dirac STUDIO and PRO (all channels are being processed at the same time). As a rule of thumb, phase is always important with stereo recordings, or recordings of the same sound source that were made simultaneously through different microphones. It is almost always the case with the channels in a surround mix. In these cases, you should use Dirac in multi-channel mode, by setting up a single Dirac object for multiple channels.

The relative phase is usually **not** important in a multi-track environment where you have different instruments on different tracks. In such cases, phase-locked processing can even be detrimental with regard to the sonic quality of the time or pitch change operation and should in general be avoided. You should create a separate Dirac object for each channel and process each channel separately via its own Dirac calls to obtain the best results.

[ü] Note that the commercial Dirac STUDIO or PRO retail version is required in order to use this feature.

With DiracLE, you can only process your material as separate channels with separate Dirac objects.

2.4 Interleaved Channel vs. Individual Channel Format[ü]

Many of today's digital audio workstations (DAWs) manage their audio data in interleaved channel format. This means one chunk of data contains all channels in sequential order instead of using a separate array dimension for each channel. Usually, this is also the format in which audio devices expect their data so this is a "natural" channel layout. On the other hand, the interleaved channel format makes it difficult to shift, cut and paste individual channels because they have to be de-interleaved and re-interleaved. In this case, it makes sense to store all channels separately and use a separate array dimension to pass the channels to Dirac for processing.

Dirac supports both channel formats, you can even use both in combination – for example you could pass the data to Dirac as individual channels and request them in interleaved format – that way you could immediately write them to a multi channel file or device.

The choice of format is made when you create your Dirac instance (`DiracCreate()`/ `DiracCreateInterleaved()` – relevant for input data format) and when you request processed data from Dirac (`DiracProcess()` / `DiracProcessInterleaved()` – relevant for output data format). Please see the next chapter for calling convention and function prototypes.

	time=0	**time=1**	**time=2**	**time=3**
channel=0	data[0][0]	data[0][1]	data[0][2]	data[0][3]
channel=1	data[1][0]	data[1][1]	data[1][2]	data[1][3]
channel=2	data[2][0]	data[2][1]	data[2][2]	data[2][3]

2.4.1 Individual channels in `data[][]` array.

	time=0	**time=1**	**time=2**	**time=3**
channel=0	data[0]	data[3]	data[6]	data[9]
channel=1	data[1]	data[4]	data[7]	data[10]
channel=2	data[2]	data[5]	data[8]	data[11]

2.4.2 Interleaved channels in `data[]` array.

2.5 Multi-Threading Safety

On modern system architectures it is sometimes desirable to use different threads for different tasks. For example, you might want to run your user interface code in a different thread than the DSP processing. Dirac is completely multi-threading safe, meaning that posting parameter changes from a different thread while processing is running is handled properly. Any of the Dirac calls can be used in an asynchronous manner with regard to the other calls. Of course, care should be taken that the Dirac object already exists before any other calls are made.

[ü] Note that this is only relevant for Dirac STUDIO and PRO. DiracLE supports only one channel per instance.

2.6 Long-Term Precision

Contrary to many other time and pitch change algorithms on the market today, Dirac has been fully tested for long-term stability and precision. This is important when time stretching extremely long musical pieces, for example an entire film score. Also, since the time and pitch change parameters are provided as long double, it is guaranteed that the conversion by a factor close to 1.0 (such as the 30/29 fps PAL/NTSC conversion) does not cause the audio track to lose sync after some time.

Note that due to its internal processing granularity Dirac might provide an output file length that is slightly different from the expected value when processing extremely short audio files.

2.7 Classic Pitch Correction vs. DiracRetune

In version 2 of Dirac we have introduced the capability to *correct* the pitch of a musically monophonic audio signal by quantizing it to the closest semitone on an equitempered scale. This capability has proven useful when treating audio tracks that were off-key. In version 3 we still support the classic pitch correction, but we also offer a new mode called DiracRetune which is more effective, less computationally demanding and stricter that our classic pitch correction algorithm. Please see the chapter on DracRetune for more details on how to set it up in your code.

Note that the classic pitch correction can use a tuning table when correcting pitch. See the relevant sections for more information on how to set this up.

3

Chapter 3: Technology Overview – APIs and Their Applications

The Dirac library contains 3 different APIs each of which is intended to be used for a particular purpose. Please find a more detailed description of their features below:

3.1.1 The Dirac Core API

The "classic" Dirac core API provides both realtime and non-realtime time stretching and pitch shifting with a variety of parameters that can be adjusted. It uses its own internal data cache which gives it the ability to process and generate data without introducing a throughput delay (processing latency). It pulls data via a callback mechanism. The callback is written and provided by the calling program in order to provide data whenever Dirac needs them to do its magic.

You should use the Dirac core API whenever:

1. …you need the best possible results for a given signal. The Dirac core API is used for mastering and post-production applications such as Prosoniq's award winning TimeFactory software
2. …you need formant corrected pitch shifting (=pitch shifting without the "munchkin effect")
3. …you need pitch correction for classical instruments or voice
4. …you need transient preserving time stretching to slow down a piece of music
5. …you need a high quality time stretching for dialogue, audio books or general narration
6. …you need full control over all aspects of the process

API Availability: DiracLE (limited feature set, single channel, 44.1/48kHz only), Dirac STUDIO, Dirac PRO, Dirac PRO mobile

3.1.2 The DiracFx API

The DiracFx API has been developed with one thing in mind: Speed. While Dirac core preview mode already gives you a very fast option to do realtime time stretching and pitch shifting, the DiracFx API delivers a process that is between 2x-4x faster. This means increased battery life on handheld devices, more tracks that can use time stretching and pitch shifting in real time, more voices on a sample player. In addition to this, DiracFx uses a constant-CPU process that utilizes about the same amount of CPU cycles no matter what settings you use, and is a very simply API with no callbacks and no extra code required for configuration. Create your DiracFx instance and process data with only 2 calls. DiracFx uses no trigonometric functions and works on 16bit integer data by design, which makes it lightning fast while keeping a low memory footprint and minimizes bus traffic and data cache use at the same time.

You should use DiracFx whenever:

1. …you need the best performance you can get
2. …you need to change pitch and speed of multiple tracks at the same time
3. …quality isn't a big issue, but performance and battery life are
4. …you want the user to preview results in real time
5. …you need to run other CPU intensive tasks, such as games

API Availability: DiracLE (single channel, 44.1/48kHz only), Dirac STUDIO, Dirac PRO, Dirac PRO mobile

3.1.3 The DiracRetune API

Many of our partners asked us for a more “strict” pitch correction, similar to what is being used in recent music productions to do vocal pitch correction, sometimes mistakenly called a “vocoder”. DiracRetune does just that. It is suited for pitch correcting voice in realtime and it creates the much sought after “pitch lock” effect that has recently become very popular. DiracRetune also allows native processing of PCM data in “short” format which allows better performance on mobile devices such as the iPhone.

You should use DiracRetune whenever:

1. …you need vocal pitch correction in real time
2. …you need a quick way to measure pitch (key within an octave and deviation from ideal tuning in cent)
3. …you wish to create artificial sounding vocal lines

API Availability: Dirac PRO, Dirac PRO mobile

3.2 The Dirac Core API - Description

The following section describes the Dirac core setup procedure and function calls.

3.2.1 DiracCreate

```
void *DiracCreate(   long lambda,
                     long quality,
                     long numChannels,
                     float sampleRateHz,
                     long (*readFromChannelsCallback)(float **data,
                            long numFrames,
                            void *userData),
                     void *userData);
```

```
void * DiracCreateInterleaved(long lambda,
                     long quality,
                     long numChannels,
                     float sampleRateHz,
                     long (*readFromInterleavedChannelsCallback)(float *data,
                            long numFrames,
                            void *userData),
                     void *userData);
```

This call allocates a new Dirac object and returns a reference to this object as a void pointer to the host program. All subsequent calls need to refer to this object by passing this void pointer as last argument in the function call. If the object could not be created, `DiracCreate()` returns a NULL pointer. Obviously, no further Dirac calls should be made in this case. After processing has completed, which the host can detect when the desired amount of output data has been produced or a desired input position has been reached, the object can be discarded using the `DiracDestroy()` call.

`DiracCreateInterleaved()` creates an instance of Dirac that expects the channel data to be in interleaved format (in the case of a stereo file, adjacent array elements would contain left, right, left, right channel data. This is a common format when dealing with multichannel sound files or realtime input from a sound card).

Note: the required amount of RAM varies with sample rate and number of channels.
Note: Please see chapter 2.4 and the below paragraph on the callback functions for details on channel data format

`lambda`
Sets the time and frequency localization tradeoff of the time-frequency basis. Please see chapter 2.2.1 for a detailed description of possible values for this parameter.

`quality`
Sets the speed/quality tradeoff of the process. Please see chapter 2.2.2 for a detailed description of possible values for this parameter.

`numChannels`[≅]
Defines the number of channels Dirac PRO should process. There is no upper limit to this number (memory permitting). Note that DiracLE will fail to create a Dirac object if you pass anything other than a value of 1. Dirac STUDIO supports only 1 and 2 channels.

[≅] Note that this is only relevant for Dirac STUDIO and PRO. DiracLE supports only one channel per instance.

`sampleRateHz`[ü]
The processing sample rate in Hertz. There is no upper limit to this number (memory permitting), the lowest possible sample rate is 8000 Hz. Note that DiracLE will fail to create a Dirac object if you pass anything other than a value of 44100 or 48000. Please see the table at the beginning of this manual for more details on sample rates supported by the different Dirac versions.

`readFromChannelsCallback`
`readFromInterleavedChannelsCallback`
A function pointer to your buffer fill function. Writing this function is up to you, since there is no way Dirac can know how your audio data and channels are organized. This function is called by Dirac whenever the library needs new input data.

It assumes that your buffer fill function `readFromChannelsCallback() / readFromInterleavedChannelsCallback()` supplies consecutive frames of data, for example, if the first call requests 10 frames and the second call requests 20 frames, it assumes that these are frames 0…9 and 10…29 in your audio channel.

`*userData`
A void* to a data structure or object that is passed to your callback procedure. It can be used to pass on an audio file object for that particular Dirac instance or any other data that you require inside that callback procedure to manage your audio channels.

Note: If you don't need this option it is recommended that you pass a `NULL` *pointer.*

[ü] DiracLE supports 44100 and 48000 Hz. Dirac STUDIO supports all sample rates from 8000-96000 Hz, Dirac PRO supports sample rates starting at 8000 Hz with no upper limit.

The Dirac Core Buffer Fill Callback Procedure

```
long readFromChannelsCallback(float **data, long numFrames, void *userData);
long readFromChannelsCallbackInterleaved(float *data, long numFrames, void *userData);
```

Note that if you create an interleaved Dirac instance using `DiracCreateInterleaved` () you have to specify a callback function with `float *data` and Dirac requires interleaved channel ordering. In this case, `data[]` must be appropriately sized to hold `numFrames*numChannels` float values.

If you create a Dirac instance that expects individual channels using `DiracCreate`() you need to supply a function pointer with a `float **data` parameter to specify individual channel ordering. See chapter 2.4 for more details on channel data ordering.

Callback Return Value
The callback should return the number of frames actually read from the file or stream, zero if none are available (EOF), or a negative error code if an error occurred. Dirac will cancel processing if a zero or negative return value is encountered.

NOTE: It is your responsibility to fill `data` *with* `numFrames` *of audio data. If your input source does not provide as many frames as requested during this call you will either need to wait inside the callback until you can successfully provide the requested frames, or return less frames (eg. when encountering EOF). In the latter case Dirac assumes a zero signal value for all remaining frames, regardless of their actual value. Dirac will always utilize all* `numFrames` *frames for its processing.*

Your callback function should accept and handle the following arguments:

`**data`
pointer to the input channel array. Your callback routine should fill the audio data of the first channel (i.e. left stereo) in `data[0][0...numFrames-1]`. The second channel should be in `data[1][0...numFrames-1]`, asf. Note that the value range for the audio data is expected to be in [-1.0, 1.0[

– or –

`*data`
pointer to the input channel data. Your callback routine should fill the audio data in interleaved format (i.e. the current frame (`frameNumber`) of the current channel (`channelNumber`) is in `data[numChannels*frameNumber+channelNumber]`. Note that the value range for the audio data is expected to be in [-1.0, 1.0[

`numFrames`
Defines the number of frames that your function must provide in `data`

NOTE: It is your responsibility to fill `data` *with* `numFrames` *of audio data. If your input source does not provide as many frames as requested during this call you will either need to wait inside the callback until you can successfully provide the requested frames, or return less frames (eg. when encountering EOF). In the latter case Dirac assumes a zero signal value for all remaining frames, regardless of their actual value. Dirac will always utilize all* `numFrames` *frames for its processing.*

`*userData`
A void* to a data structure or object that will be passed on to your callback procedure when it is called by `DiracProcess()`. It can be used to pass on an audio file object for that particular Dirac instance or any other data that you require inside that callback procedure to manage your audio channels.

If you don't need the `*userData` *option it is recommended that you pass a* `NULL` *pointer.*

3.2.1.1 Important Notes on Callback Function Operation

There is no way to actually predict how many frames Dirac needs per call and there is no guarantee that Dirac requests an equal amount of frames at each consecutive call.

Please note also that the number of frames requested through your callback function can depend on a variety of factors, such as sample rate, Dirac library version and selected Dirac quality mode. You should therefore not make any assumptions about these figures, and take care that your callback function will handle all non-serviceable requests gracefully.

In particular, since Dirac has no idea of how your audio channels are laid out and how much data they contain, make sure you catch any end-of-file condition that may arise from requesting data through your callback function.

Should Dirac request nonexistent data, make sure you pass zero'ed frames instead.

The callback function takes an optional void pointer that you can use to pass on any information you need (such as a reference to an object or file) to the code inside that function.

The property `kDiracPropertyCacheNumFramesLeftInCache` can be used to make an estimate when Dirac is going to invoke the callback to ask for more data, so you can avoid calling `DiracProcess` unless you have enough data available.

It is safe to make get/set property calls to Dirac from within the callback.

3.2.1.2 Seeking in a File During Processing

It is possible to interrupt processing and seek to a specific location in a file. Since the caller (you) is responsible for keeping track of the input position in the file you can jump to a different location at any time during processing. A few things need to be considered when doing this, however:

1. We recommend that you stop processing while the seek is underway if you're executing `DiracProcess()` on a separate thread
2. Once you're done seeking, and immediately before you resume processing, you need to call `DiracReset()` in order to re-init the Dirac data cache.

3.2.1.3 Reversing Playback Direction

Dirac relies on you to supply contiguous data when the callback is invoked. However, Dirac does not require that data to come from a particular position in your file so it is entirely up to you to create and manage that data. This gives you the freedom to preprocess your audio signal or change playback direction by reading from the end of the file and reverse the order of the frames in the buffer before passing it back to Dirac.

However, many codecs (such as MP3) depend on time being continuous when decoding a file, specifically, they use decoding information from past frames to update internal state variables in order to decode more recent frames. In this case, reading from the end of a file involves a costly seek operation to the frames just before the position that you want to decode, and a subsequent decoding process that involves a significantly larger part of the file than you have actually requested. In some cases, the seek process even has to start from the beginning of the file each time you read a chunk out of sequence, which totally ruins performance. Therefore, we recommend you decode your files into PCM format prior to using any calls that access audio data out of sequence.

3.2.1.4 Testing your Dirac Core Implementation

We generally recommend using one of our example projects as a template when starting out with integrating Dirac into your own projects. However, in some cases this might not be feasible. In these cases it can be beneficial to have a way to test your Dirac implementation without actually calling Dirac.

In this case we recommend replacing the DiracProcess() call directly with the callback that you have implemented. When set up like this, the entire code can be debugged since it is not actually executing anything hidden away inside our library. So instead of making a call like this

```
long ret = DiracProcess(bufferAddress, bufferNumberFrames, mDirac);
```

you change the call to look like this:

```
long ret = myReadData(bufferAddress, bufferNumberFrames, this);
```

This should work as a passthrough replacement for the actual DiracProcess/DiracProcessInterleaved call and enable you to test and debug your implementation.

3.2.2 DiracSetProperty and DiracGetProperty

```
long DiracSetProperty(long selector, long double value, void *Dirac);
long double DiracGetProperty(long selector, void *Dirac);
```

`DiracSetProperty()` is used to actually configure the Dirac algorithm before (or during[ü]) processing. It has a variety of selectors that you can use to set a specific property at any given time. Please refer to the below table for a detailed list of selectors and a description of their effect.

The corresponding function `DiracGetProperty()` can be used to determine the value of a selected parameter at any given time. Following are the most important and frequently used selectors along with an explanation of their effect:

Selector	Range	Description	Note
`kDiracPropertyPitchFactor`	0.5 – 2.0	Set/get desired/current pitch shift factor (1.0=original, 0.5=octave down, 2.0=octave up)	
`kDiracPropertyTimeFactor`	0.5 – 2.0	Set/get desired/current time stretch factor (1.0=normal, 0.5=double speed, 2.0=half speed)	
`kDiracPropertyFormantFactor`	0.5 – 2.0 / 0.0, 1.0	Set/get desired/current formant scale factor. Dirac PRO: Typically 1./pitch for natural pitch shifting, 1.0=regular pitch shifting with formant shift	
`kDiracPropertyCompactSupport`	0.0, 1.0	Set/get compact support for the time-frequency decomposition. 1.0 enables compact support, 0.0 disables it. Compact support refers to the basic time frequency decomposition and has only minor bearing on the acoustic outcome of the process. If the result sounds too coarse for all lambda settings you can try setting this parameter to 0.0 for a slightly smoother result.	If not set this is assumed to be 1.0
`kDiracPropertyCacheGranularity`	0 – 4096	Set/get desired/current read chunk size of the input cache. Smaller values will result in more read calls with less frames per read request, larger values will need the callback fewer times per second but will request more frames during each call.	Default is 4096
`kDiracPropertyCacheMaxSizeFrames`	arbitrary	Get current read cache size in frames. This is the size of the internal cache that holds audio data. Dirac will never read more frames from its input than this.	Read only property
`kDiracPropertyCacheNumFramesLeftInCache`	arbitrary	Gets the number of frames that are currently available in Dirac's input buffer. If this number is close to zero Dirac will use its callback to get more data. This is a handy read-only property to determine whether or not it is safe to call DiracProcess() with the amount of data available, for	Read only property

[ü] Note that the commercial Dirac STUDIO or PRO retail version is required in order to use this feature.

		instance when reading from a stream that might not always be ready to supply data when required.	
`kDiracPropertyUseConstantCpuPitchShift`	0.0, 1.0	A value of 1 enables constant CPU load pitch shifting. This mode causes all pitch shift factors to require approximately the same amount of CPU cycles. In previous versions, raising the pitch would increase CPU load proportionally. This affects all modes that support pitch shifting (kDiracLambda1-5, kDiracTranscribe).	This property has some impact on quality when upshifting pitch so it is off by default and its use should be strictly limited to realtime processing/previewing. **NOTE: Recommended to be enabled only when upshifting pitch**
`kDiracPropertyDoPitchCorrection`	0.0, 1.0	Set/get desired/current status for the pitch correction option. If set to 1, pitch correction is enabled and will cause the pitch of the input signal to be quantized to discrete notes	This enables classic pitch correction (see 2.7 for more details). This feature requires that the Dirac lambda value is set to `kDiracLambda1` (musically monophonic instruments and voice). Note that this mode disables time stretching and pitch shifting.
`kDiracPropertyOutputGainDb`	-90.0 – 24.0	Set/get desired/current output gain that is to be applied to all audio going through this instance of Dirac. You can use this to change the volume on a particular Dirac instance when mixing tracks.	Number is specified in decibels (dB). A gain of –6.02dB reduces the overall signal volume to half, a gain of +6.02dB doubles volume.
`kDiracPropertyPitchCorrectionBasicTuningHz`	400.0 – 500.0	Set/get desired/current reference tuning (in Hz) for the pitch correction. Default is 440 Hz	This feature requires that the Dirac lambda value is set to `kDiracLambda1` (musically monophonic instruments and voice) and `kDiracPropertyDoPitchCorrection` is enabled.
`kDiracPropertyPitchCorrectionDoFormantCorrection`	0.0, 1.0	If set to 1 this option causes the pitch shift to keep the formants in place, making the change sound more natural (PRO version only).	This feature requires that the Dirac lambda value is set to `kDiracLambda1` (musically monophonic instruments and voice) and `kDiracPropertyDoPitchCorrection` is enabled.
`kDiracPropertyPitchCorrectionSlurTime`	0.0 – 20.0	Set/get the time it takes for the correction to reach the full correction amount. Typically, notes are a bit unstable at the beginning, because the attack phase of a sound has a higher amount of noise, and because singers gradually adjust their tuning after the onset of the note. The slur time makes the pitch correction sound natural because it models this effect. Higher values will yield a slower adaptation time and it will take longer for the correction to produce the corrected pitch. However, longer slur times will also preserve vibrato better.	This feature requires that the Dirac lambda value is set to `kDiracLambda1` (musically monophonic instruments and voice) and `kDiracPropertyDoPitchCorrection` is enabled.
`kDiracPropertyPitchCorrectionFundamentalFrequency`	-1 (off), 0.0 (no detected	Set/get the fundamental frequency of the currently processed audio signal.	This feature requires that the Dirac lambda value is set to

	pitch), 50 – 1000 Hz	This can be used to analyze and/or control the pitch of the current note via MIDI.	`kDiracLambda1` (musically monophonic instruments and voice) and `kDiracPropertyDoPitchCorrection` is enabled. **Experimental feature**

Note that getting and setting the DiracRetune properties from the Dirac core API has been discontinued in version 3.5. See the section on the DiracRetune API for more information.

`selector`
Defines the property to be examined/set. The available properties are defined as enum constants in Dirac.h.

`*value`
Pointer to a variable of type long double that contains/should contain the actual value of the parameter to be passed/retrieved

`*dirac`
void pointer to the allocated Dirac object

Note that for DiracLE setting parameters with `DiracSetProperty()` *does have no effect once processing has been started.*

Return value
`DiracSetProperty()` returns `kDiracErrorNoErr` when the parameter change has been accepted and `kDiracErrorParamErr` if an error occurred. If returms `kDiracErrorFeatureNotSupported` if the option you're trying to use is not available in the current context or Dirac license model.

`DiracGetProperty()` returns the value for the selected parameter or 0 if an error occurred.

3.2.2.1 Detailed Description of Time, Pitch and Formant Change Operation

`kDiracPropertyTimeFactor`
Determines the amount of time scaling applied. Values can range from 0.5 to 2.0, where a value of 1.0 means no speed change, 0.5 will shrink the duration of the data set to be half the original length (making it play back twice as fast without changing its pitch), 2.0 will double the playback length, making it twice as slow without changing its pitch.

Hint: If you wish to apply a speed change that also affects pitch (a *Pitch Transpose Effect*), you should set `pitchScale` and `timeScale` as follows:

```
            pitchScale = pow(2., semitone/12. + cent/1200.);
            timeScale = pow(2., -semitone/12. - cent/1200.);
```

`kDiracPropertyPitchFactor`
This selector determines the amount of pitch shifting that should be applied. Values range from 0.5 to 2.0, corresponding to a total pitch shifting range of 2 octaves, where a value of 1.0 means no pitch change, 0.5 will transpose the data set down by one octave without changing its speed, 2.0 will double the pitch, making it one octave higher - again without affecting its playback speed. The pitch shifting value may be easily calculated from the semitone and cent values set by the user by

```
            pitchShift = pow(2., semitone/12. + cent/1200.);
```

Note that by default Dirac uses a high quality pitch shifting mode that requires more CPU cycles as the pitch of the signal gets shifted up. If you are using Dirac in a realtime processing environment and are increasing pitch we recommend you set the property `kDiracPropertyUseConstantCpuPitchShift` *to switch to a processing mode that requires approximately the same amount of computations for all pitch shift factors.*

`kDiracPropertyFormantFactor`
Defines the amount of formant scaling that is applied. Dirac allows an arbitrary formant factor, allowing for gender transformation and voice post-processing in addition to achieving a natural pitch shift. The value can range from 0.5 to 2.0 with 0.5 scaling the formants down by one octave and 2.0 scaling the formants up one octave. 1.0 has no effect and disables this feature. To maintain natural transposition or pitch shifting, the `formantScale` value should be calculated from the pitch shifting values as follows:

```
            formantScale = pow(2., -semitone/12. - cent/1200.);
```

Note that you can scale the formants without changing the pitch, thereby achieving interesting voice transformation effects.

Note that formant processing should be off (set to 1.0) in a realtime context as it requires a lot of additional cycles.

3.2.2.2 Using Dirac for Classic Pitch Correction

Automatic detection and correction of pitch (intonation) has become more and more important over the recent years. It has been used as a special effect to produce synthetic sounding voice and it has been used to correct the intonation of a singer or instrumentalist to be more precise. Both can be achieved with all versions of Dirac, however, we tried to implement this feature in a more natural sounding way, preserving the original tone and quality of the recording.

Pitch Correction is enabled by calling `DiracSetProperty` with the selector `kDiracPropertyDoPitchCorrection` and the value 1 before (or during[ü]) processing.

NOTE that if pitch correction is enabled, the pitch and formant shift parameters are ignored.

NOTE that you must create your Dirac instance with a lambda setting of 1 (`kDiracLambda1`*) in order for the pitch correction to work. If you don't do this pitch correction will be disabled.*

NOTE that pitch correction works best for recordings that have a single fundamental frequency, such as voice or single instruments. If you apply pitch correction to entire mixes you might get strange results.

There are several parameters that can be set through calls to `DiracSetProperty` with the below selectors before (or during[ü]) processing. See section 3.2.2 for a complete list and allowed value ranges.

`kDiracPropertyDoPitchCorrection`
Specifies the status for the pitch correction option. If set to 1, pitch correction is enabled and will cause the pitch of the input signal to be quantized to discrete notes.

`kDiracPropertyPitchCorrectionBasicTuningHz`
Defines the reference tuning (in Hz) for the pitch correction. Default is 440 Hz.

[ü] Note that the commercial Dirac STUDIO or PRO retail version is required in order to use this feature.
[ü] Note that the commercial Dirac STUDIO or PRO retail version is required in order to use this feature.

`kDiracPropertyPitchCorrectionSlurTime`
Defines the time it takes for the correction to reach the full correction amount. Typically, notes are a bit unstable at the beginning, because the attack phase of a sound has a higher amount of noise, and because singers gradually adjust their tuning after the onset of the note. The slur time makes the pitch correction sound natural because it models this effect. Higher values will yield a slower adaptation time and it will take longer for the correction to produce the corrected pitch. However, longer slur times will also preserve vibrato better.

```
long DiracSetTuningTable(float *frequencyTable, long numFrequencies, void *dirac);
```

Sets a tuning table for the classic pitch correction. Note that a typical size is 88 keys. Larger tables are possible but will be inefficient. The table entries do not have to be in ascending order of frequency.

A setting < 0 for `numFrequencies` causes processing to switch correction off.

3.2.3 DiracProcess

```
long DiracProcess(float **data, long numFrames, void *dirac);
long DiracProcessInterleaved(float *data, long numFrames, void *dirac);
```

When you're done setting up your Dirac object you should enter the main processing loop. This is done by simply calling the Dirac library function `DiracProcess()` periodically, checking its return value to determine whether all data have been processed.

`DiracProcess()` returns the number of frames in the `data` array (ie. the returned value should be `numFrames` during processing), or 0 when no further processed data are available.

Note that if you require the output data to be in interleaved channel format you can also call the interleaved version `DiracProcessInterleaved()`. You do not have to create an interleaved instance of Dirac to request the *output* data in interleaved format.

`**data`
`*data`
pointer to the output channel array.

If multi-dimensional, the first channel (i.e. left stereo) is expected to be in `data[0][0...numFrames-1]`, where `numFrames` is your value for the `data[n][...]` array size passed to this function. The second channel will be in `data[1][0...numFrames-1]`, asf.

If one-dimensional, the output channels are returned in an interleaved manner. See chapter 2.4 for details on channel data ordering.

Note that DiracLE supports only one audio channel per instance.

Note that the value range for the audio data is expected to be in the range [-1.0, 1.0[

Upon return, `data[][]` contains the output channel data. Note that the input data is not provided via this function, it is requested from your application via the callback function defined in `DiracCreate()` or `DiracCreateInterleaved()`.

`numFrames`
The number of output frames you want to have upon return. There are no restrictions to this number, but make sure you allocate a large enough `data[][]` array to hold the output signal. Also, since the call to this function is synchronous, your program (thread) will be unresponsive if you use a large value for `numFrames` while Dirac is producing the requested amount of data frames.

`*dirac`
void pointer to the allocated Dirac object.

Please see the accompanying example project for more information

3.2.4 DiracDestroy

```
void DiracDestroy(void *dirac);
```

When you're done processing your audio data with Dirac, you should free the memory used by Dirac. This is done using a single call to `DiracDestroy`. `DiracDestroy` destroys a Dirac object and frees all its associated memory.

`*dirac`
void pointer to the allocated Dirac object.

3.2.5 DiracReset

```
void DiracReset(bool clear, void *dirac);
```

`DiracReset` resets a Dirac object optionally filling all its internal buffers with zeros (`clear` is true). Resetting should be used if you wish to process different parts in a file with the same settings and don't want the previous signal to "spill" into the next segment. If you want the output to be contiguous without having an audible gap you should pass "false" in `clear`.

You need to call `DiracReset` immediately before resuming processing if you are seeking to a specific location in a file during processing (see section 3.2.1.2 for more information)

`clear`
Set to `true` if you want Dirac to clear its internal buffers.

`*dirac`
void pointer to the allocated Dirac object.

3.2.6 DiracSetProcessingBeganCallback[3]

```
void DiracSetProcessingBeganCallback(void (*processingCallback)(unsigned long position, void
*userData), void *userData, void *dirac);
```

`DiracSetProcessingBeganCallback` gives you an opportunity to install a callback directly into Dirac's processing core that gets invoked whenever Dirac processes a single block of data. This can be used for a more fine-grained control over Dirac's properties, as you can call `DiracSetProperty()` from within that callback to change Dirac's behaviour. `processingCallback` callback is invoked immediately prior to Dirac's DSP core, so any change you'll be making in the callback will be used immediately for the data at the specified input frame position.

EXAMPLE: First you will want to write your own callback:

```
void processingCallback(unsigned long inputFramePosition, void *userData)
{
  void *dirac = userData;

  // As an example, set time stretch to 200% after one second @ 44.1kHz
  // making sure we don't apply time stretch otherwise
  if (inputFramePosition >= 44100.f)
        DiracSetProperty(kDiracPropertyPitchFactor, 2.0, dirac);
  else
        DiracSetProperty(kDiracPropertyPitchFactor, 1.0, dirac);


}
```

Now set up Dirac and install the callback:

```
void *dirac = DiracCreate(kDiracLambdaPreview, kDiracQualityPreview, numChannels, \
                          sampleRate, &myReadData, (void*)&state);
if (!dirac) {
  printf("!! ERROR !!\n\n\tCould not create DIRAC instance\n");
```

[3] Only available in Dirac PRO and Dirac PRO mobile

```
        exit(-1);
    }

    DiracSetProcessingBeganCallback(&processingCallback, dirac, dirac);
```

Once you've done this continue with your program calling `DiracProcess()` repeatedly until processing has ended. Don't forget to include your regular data provider callback `myReadData` in the code as well.

In this particular case you're passing your Dirac instance pointer to the callback so you can use `DiracSetProperty()` to change its parameters. Typically you will want to pass a pointer to your own class, and make your Dirac instance a member of that class so you can access it from within the callback while at the same time having access to your own class members.

NOTE: passing NULL in `processingCallback` deinstalls the callback.

3.3 The DiracRetune API - Description

DiracRetune is an addition to the existing classic pitch correction algorithm that was introduced in version 2. While the classic algorithm is suited for subtle pitch correction on a variety of sound sources, DiracRetune has been developed to be used specifically with vocal recordings and can be used to create synthetic sounding vocal pitch correction by forcing the input pitch to a given key.

PLEASE NOTE: DiracRetune and the DiracRetune API properties are not available in DiracLE and STUDIO.

3.3.1 Creating a DiracRetune Instance

You create a DiracRetune instance by calling `DiracRetuneCreate.`

Please note the following restriction:

1. DiracRetune operates on single channel audio only

The following calls are available:

```
void *DiracRetuneCreate(long quality, float sampleRateHz, float referenceTuningHz);
```

Creates a DiracRetune instance and returns the created object. Please note that DiracRetune only supports single channel audio. `referenceTuningHz` should reflect the desired reference tuning (usually this is A = 440Hz). Pass 440. if you're unsure which value to use.

Use one of the following selectors as quality parameter:

Value	**Description**
`kDiracQualityPreview`	This quality mode offers preview quality, which is usually good enough for a preview to see the effects of the parameter settings or for realtime processing.
`kDiracQualityGood`	A better quality mode than `kDiracQualityPreview`. It is recommended as the default quality setting for non-realtime (non-preview) processing
`kDiracQualitBetter`	Very good quality mode but takes more CPU.
`kDiracQualityBest`	The highest quality mode. Note that this setting can be slow.

3.3.2 Destroying DiracRetune

```
void DiracRetuneDestroy(void *diracRetune);
```

Destroys a DiracRetune instance. Note that you need to destroy a DiracRetune instance with a call to `DiracRetuneDestroy` and NOT `DiracDestroy.`

3.3.3 Processing

```
void DiracRetuneProcess(short *audioIn, short *audioOut, long numFrames, void *diracRetune);
```

Processes a chunk of short data out of place. Note that you can also use the same pointer as indata and outdata but this will be less efficient.

```
void DiracRetuneProcessFloat(float *indata, float *outdata, long numSampsToProcess, void
*instance);
```

Processes a chunk of float data out of place. Note that you can also use the same pointer as indata and outdata but this will be less efficient. In general, the short data call is more efficient than the float variant, especially on mobile devices such as the iPhone.

3.3.4 Configuring DiracRetune, Utility Functions

```
void DiracRetuneSetProperties(float correctionAmountPercent,
                              float correctionCaptureCent,
                              float correctionAutoBypassThreshold,
                              float correctionAmbienceThreshold,
                              void *instance);
```

Sets the the properties for DiracRetune.

`correctionAmountPercent` `(0...100)`
Defines the amount of the correction applied. 0 means no correction, 100 means full correction

`correctionCaptureCent` `(0...99)`
Defines the threshold at which the correction takes effect. For most applications this should be set to 0 as this will correct notes that are more that 0 cent off, wrt their ideal pitch. A setting of 10 will allow a pitch deviation of 10 cents in either direction before the correction takes effect.

`correctionAutoBypassThreshold` `(0...500)`
Defines the sibilance threshold at which the original signal is passed through unaffected. This bypass switch is required to make sure that sibilancies in vocal recordings don't sound too processed. A default setting of 100 is a good tradeoff.

`correctionAmbienceThreshold` `(-10...10)`
Sets the automatic bypass when the signal contains too much noise or background ambience (reverb). A value between 0.2 and 0.4 is usually a good starting point

```
float DiracRetuneGetPitchHz(void *instance);
```

Returns the average pitch of the segment that is currently being analyzed. This can be used as a pitch detector. A return value of 0.0 means that no coherent pitch could be estimated. You should expect an output fundamental frequency in the range of 50 Hz to 1000 Hz.

Please note that this feature is experimental and might produce unpredictable results. In particular, it might introduce octave errors for the measured pitch depending on the input signal.

```
void DiracRetuneSetKeyList(float *tuningCentRelativeToKey0, long numKeysPerOctave,
                           long octaveOffsetKeyNo, void *diracRetune);
```

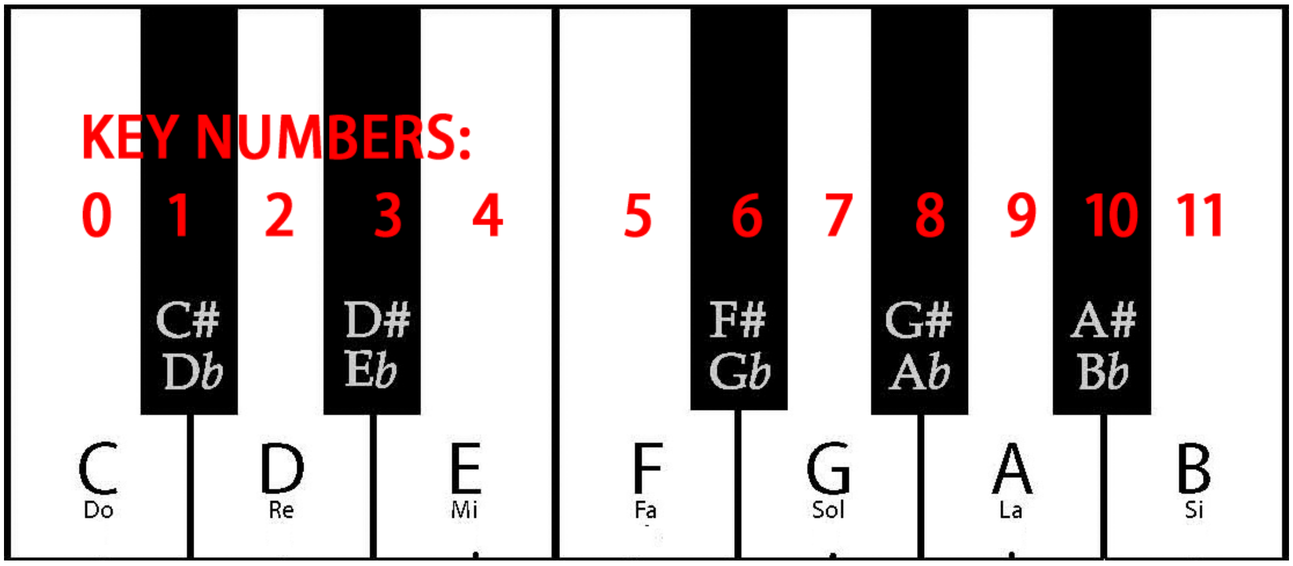
This call can be used to fine tune the individual notes of an octave. Tuning is specified relative to key 0 (the lowest key in the octave, usually C) in cents (100 cents = 1 semitone). An octave may be larger or smaller than 12 keys, and the entire keyboard may start with any key specified as an offset with `octaveOffsetKeyNo`.

The following key list will create an equitempered scale of 88 keys (a piano keyboard):

```
    float kl[] = {0.f,100.f,200.f,300.f,400.f,500.f,600.f,700.f,800.f,900.f,1000.f,1100.f};
    DiracRetuneSetKeyList(kl, 12, 3, diracRetune);
```

An offset of 3 is specified because the keys on a piano start out with A0, hence the first C (C1, our key 0) is 3 semitones up.

Note that calling `DiracRetuneSetKeyList()` *resets the key state obtained through* `DiracRetuneGetKeyStatus()`. *You will have to set the key status again after setting a new key list.*



```
bool DiracRetuneGetKeyStatus(long keyNo, void *instance);
void DiracRetuneSetKeyStatus(long keyNo, bool enable, void *instance);
```

Sets or obtains the status (on = true, off = false) for any given key in the octave. Usually, keyNo will be in the range 0…11, but if you're using a key list with a larger or smaller octave this may be any key in your octave.

Disabling a key will cause the pitch correction to ignore it, and round the pitch up or down to the closest key that is adjacent to the omitted key.

```
unsigned long DiracRetuneGetAllowedKeysMask(void *instance);
void DiracRetuneSetAllowedKeysMask(unsigned long mask, void *instance);
```

Allows manipulating the key mask directly. Bit position in mask determines the key that is to be switched on (1) or off (0). This simplifies storage and retrieval of the key settings without repeated calls to `DiracRetuneGetKeyStatus/DiracRetuneSetKeyStatus`. Note that all keys are enabled by default.

NOTE: The actual word length of the `unsigned long` *data type may vary with platform and target settings. Make sure you are aware of this and the restrictions it might impose on your octave sizes.*

```
long DiracRetuneGetClosestKey(bool respectKeyState, void *instance);
```

Returns they key number in your octave that is closest to the measured pitch at the time this call is made. Usually, the returned key number will be in the range 0…11, but if you're using a key list with a larger or smaller octave this may be any key in your octave. Note that if the pitch cannot be determined, –1 is returned instead. Does respect the key state (set with `DiracRetuneSetKeyStatus, DiracRetuneSetAllowedKeysMask`) if `respectKeyState` is true;

```
float DiracRetuneGetClosestKeyDetuneCent(bool respectKeyState, void *instance);
```

Returns a measurement by how much the current pitch is off from the correct pitch (determined by `DiracRetuneGetClosestKey`) in cent. Does respect the key state (set with `DiracRetuneSetKeyStatus, DiracRetuneSetAllowedKeysMask`) if `respectKeyState` is true.

```
void DiracRetunePrintInternalTuningTable(void *instance)
```

Prints a list of the tuning table that is used internally to stdout. It prints the key number, tuning in Hz, enabled/disabled state (an "X" denoting enabled state) and the tuning table value in cent. Octaves are separated by dotted lines. This is mainly a debugging tool. Example output is shown below.

```
Printing DiracRetune tuning table (88 keys):

    [key #  0]          [27.5000000000 Hz]       [ ]        [900.00 cent]
    [key #  1]          [29.1352348328 Hz]       [ ]        [1000.00 cent]
    [key #  2]          [30.8677062988 Hz]       [ ]        [1100.00 cent]
  ...................................................................... < OCT
    [key #  3]          [32.7031974792 Hz]       [X]        [0.00 cent]
    [key #  4]          [34.6478271484 Hz]       [ ]        [100.00 cent]
    [key #  5]          [36.7080955505 Hz]       [ ]        [200.00 cent]
    [key #  6]          [38.8908729553 Hz]       [X]        [300.00 cent]
    [key #  7]          [41.2034454346 Hz]       [ ]        [400.00 cent]
    [key #  8]          [43.6535301208 Hz]       [ ]        [500.00 cent]
    [key #  9]          [46.2493019104 Hz]       [ ]        [600.00 cent]
    [key # 10]          [48.9994277954 Hz]       [X]        [700.00 cent]
    [key # 11]          [51.9130859375 Hz]       [ ]        [800.00 cent]
    [key # 12]          [55.0000000000 Hz]       [ ]        [900.00 cent]
    [key # 13]          [58.2704696655 Hz]       [ ]        [1000.00 cent]
    [key # 14]          [61.7354125977 Hz]       [ ]        [1100.00 cent]
  ...................................................................... < OCT
    [key # 15]          [65.4063949585 Hz]       [X]        [0.00 cent]
    [key # 16]          [69.2956542969 Hz]       [ ]        [100.00 cent]
    [key # 17]          [73.4161911011 Hz]       [ ]        [200.00 cent]
       .                      .                    .              .
       .                      .                    .              .
       .                      .                    .              .
```

```
long DiracRetuneLatencyFrames(float sampleRateHz)
```

Returns they number of frames by which the output will lag, given the sample rate specified.

3.3.5 Deprecated DiracRetune Calls

These calls are still available in v3.5 but their use is discouraged as they will likely go away in the near future. Feedback from our users shows that they are rarely used. Should you absolutely require one of these calls please contact us.

```
[DEPRECATED]
void DiracRetuneSetPitchHz(float pitchHz, void *instance);
```

When this call is used, DiracRetune forces the input signal to the pitch set with pitchHz. You can use this call to control the pitch shifted signal programmatically or externally, eg. via a MIDI keyboard. Set pitchHz to a value of –1 to disable this feature if you want DiracRetune to revert to using the standard pitch table instead.

We recommend you discontinue using this call and change to `DiracRetuneSetKeyStatus` instead.

Please note that this feature is currently experimental and might produce unpredictable results.

```
[DEPRECATED]
void DiracRetuneSetTuningTable(float *frequencyTable, long numFrequencies, void *diracRetune);
```

Sets a tuning table for DiracRetune. Note that a typical size is 88 keys. Larger tables are possible but will be inefficient. The table is expected to contain a list of allowed frequencies that the pitch should be quantized to.

A setting < 0 for `numFrequencies` causes processing to switch correction off.

This call has been replaced by `DiracRetuneSetKeyList()`.

```
[DEPRECATED]
void DiracRetuneSetTuningReferenceHz(float referenceTuningHz, void *diracRetune);
```

Sets the basic tuning reference frequency. This defaults to 440 Hz. This call has no effect in v3.5+

3.4 The DiracFx API - Description

The DiracFx API has been developed with one thing in mind: Speed. While Dirac core preview mode already gives you a very fast option to do realtime time stretching and pitch shifting, the DiracFx API delivers a process that is between 2x-4x faster. This means increased battery life on handheld devices, more tracks that can use time stretching and pitch shifting in real time, more voices on a sample player. In addition to this, DiracFx uses a constant-CPU process that utilizes about the same amount of CPU cycles no matter what settings you use, and is a very simply API with no callbacks and no extra code required for configuration. Create your DiracFx instance and process data with only 2 calls. DiracFx uses no trigonometric functions and works on 16bit integer data by design, which makes it lightning fast while keeping a low memory footprint and minimizes bus traffic and data cache use at the same time.

PLEASE NOTE: DiracFx supports only single channel 44.1 and 48kHz audio in DiracLE.

3.4.1 Creating a DiracFx Instance

```
void *DiracFxCreate(long quality, float sampleRateHz, long numChannels);
```

Creates a new DiracFx instance. Note that sample rates other than 44.1/48kHz and channel counts other than 1 will cause a nil object to be returned in DiracLE.

See section 2.2.2 for a detailed description of the quality parameter.

Using multiple channels with DiracLE is possible by allocating a separate DiracFx instance for each channel, however, please note that stereo phase lock will not be established between channels. If you are processing a stereo mix this can result in a distorted stereo image and cancellations when switching to mono. Make sure you use our phase locked stereo version that comes with the STUDIO and PRO licenses.

Time stretching in DiracFx is done by performing a pitch shift followed by a sample rate conversion in order to change the length of the segment. This means that the more you slow down the signal, the less fidelity you will get since the bandwidth of the resulting signal is also reduced. If you need a higher quality we recommend you upsample the signal by a factor as large as your intended maximum time stretch factor before applying DiracFx. That way the audio fidelity will be maintained (at the expense of requiring more cycles to execute, of course).

3.4.2 Destroying DiracFx

```
void DiracFxDestroy(void *instance);
```

Destroys a DiracFx instance. Note that you need to destroy a DiracFx instance with a call to `DiracFxDestroy` and NOT `DiracDestroy`!

3.4.3 Processing

```
long DiracFxProcessFloat(long double timeFactor, long double pitchFactor,
                            float **indata, float **outdata, long numInputFrames,
                            void *instance);
```

```
long DiracFxProcessFloatInterleaved(long double timeFactor, long double pitchFactor,
                            float *indata, float *outdata, long numInputFrames,
                            void *instance);
```

```
long DiracFxProcess(long double timeFactor, long double pitchFactor,
                            short **indata, short **outdata, long numInputFrames,
                            void *instance);
```

```
long DiracFxProcessInterleaved(long double timeFactor, long double pitchFactor,
                            short *indata, short *outdata, long numInputFrames,
                            void *instance);
```

Processes a chunk of data in the specified format out of place. Note that as a rule of thumb you can **NOT** use the same pointer for `indata` and `outdata` because the output might contain less or more sample frames than the input. DiracFx issues a warning to this effect and returns without processing anything if `indata == outdata.` Note that as an exception to this rule you can use the same buffer if you're only doing pitch shifting, where input and output data sizes remain constant throughout the process. In the case that `timeFactor == 1.0`, in-place processing is permitted.

In general, the non-interleaved short data call is more efficient than the other variants, especially on mobile devices such as the iPhone.

In order to find out about the maximum size required for the output array you can make a call to `DiracFxMaxOutputBufferFramesRequired()` with the respective timeFactor and pitchFactor values, or allocate an array of sufficient size (using the maximum timeFactor value that you're going to allow) before starting processing. Note that this is the maximum size required to hold the processed data, the actual amount of data returned by `DiracFxProcessXXX` can be smaller than this number.

If you need to know exactly how many frames DiracFx is going to return by the end of the next processing call use `DiracFxOutputBufferFramesRequiredNextCall()`. This call returns the number of frames produced by DiracFx given the current time and pitch settings. Note that this number is always equal to or smaller than the number returned by `DiracFxMaxOutputBufferFramesRequired()`.

All variants of `DiracFxProcessXXX` return the amount of frames stored in `outdata` upon return. Note that the returned value might differ from the value reported by `DiracFxMaxOutputBufferFramesRequired()` depending on the size of the buffer and its relation to sample rate and time stretch factor. Specifically, the returned value will always be equal to or smaller than the value reported by `DiracFxMaxOutputBufferFramesRequired()`.

Make sure you only use as many frames from the output buffer as indicated by the returned value from `DiracFxProcessXXX`, and **not** the actual size allocated.

Note also that DiracFx introduces processing delay (latency) into your signal. The DSP passthrough latency in frames can be estimated using the `DiracFxLatencyFrames()` API call. You can then discard the initial `DiracFxLatencyFrames()` frames, and append them at the end of the input stream as zero frames in order to get the exact amount of frames back.

Note that if you apply time stretching the latency will also undergo time stretching. If the reported latency is 1,000 frames, the number of frames before the onset of the signal will be 1,500 if you're applying 150% time stretching.

3.4.4 DiracFx Utility Functions

```
long DiracFxMaxOutputBufferFramesRequired(long double timeFactor, long double pitchFactor,
                                           long numInputFrames);
long DiracFxOutputBufferFramesRequiredNextCall(long double timeFactor, long double pitchFactor,
                                           long numInputFrames, void *instance);
```

```
long DiracFxLatencyFrames(float sampleRate);
```

DiracFx utility functions – see description under `DiracFxProcessXXX`.

4

Chapter 4: Getting Results Quickly – Using DiracFx

The following example program demonstrates how to quickly set up and run DiracFx. It is also included with the Dirac library in the example projects folder. See the comments in the code for more information.

```
/*
 DiracFx "main.cpp" Example Source File - Disclaimer:
 Copyright © 2005-2012 Stephan M. Bernsee, http://www.dspdimension.com. All Rights Reserved

 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "MiniAiff.h"
#include "Dirac.h"

int main()
{

     /* ***************** SET UP ******************* */
     /* First we set our time an pitch manipulation values */
     float time      = 1.15;                               // 115% length
     float pitch     = pow(2., 3./12.);   // pitch shift (3 semitones)

     /* set up our input/output files and retrieve */
     /* sample rate, length and number of channels */
     /* Note that DiracLE supports only one channel! */
#ifdef __APPLE__
     char infileName[]="../../test.aif";
#else
     char infileName[]="test.aif";
#endif
     char oufileName[]="out.aiff";

     /* get file info */
     long numChannels    = mAiffGetNumberOfChannels(infileName);
     float sampleRate    = mAiffGetSampleRate(infileName);
     unsigned long inputNumFrames = mAiffGetNumberOfFrames(infileName);
     if (sampleRate <= 0.f) {printf("Error opening input file\n"); exit(-1);}

     /* Instantiate our DiracFx object */
     void *diracFx = DiracFxCreate(kDiracQualityBetter, sampleRate, numChannels);
     if (!diracFx) {
        printf("!! ERROR !!\n\n\tCould not create DiracFx instance\n");
        exit(-1);
     }

     /* Initialize our output file */
     mAiffInitFile(oufileName, sampleRate /* sample rate */, 16 /* bits */, numChannels);

     /* Print version info to stdout */
     printf("Running DIRAC version %s\nStarting processing\n", DiracVersion());

     /* Keep track of how many frames we've already processed */
     unsigned long inputFramesProcessed = 0;

     /* This is the amount of frames per read operation */
     /* It is an arbitrary number of frames. Change as you see fit */
```

```
      long numFrames = 8192;

      /* Allocate buffer for output */
      float **audioIn = mAiffAllocateAudioBuffer(numChannels, numFrames);
      float **audioOut = mAiffAllocateAudioBuffer(numChannels,
                                  DiracFxMaxOutputBufferFramesRequired(time, pitch, numFrames));

      /* ***************** HANDLE LATENCY ******************* */
      /* Get latency estimate */
      long latencyFrames = DiracFxLatencyFrames(sampleRate);

      /* Establish a separate buffer to account for latency. */
      /* We could do this with our processing buffers but we're lazy. */
      float **latencyBufferIn = mAiffAllocateAudioBuffer(numChannels, latencyFrames);
      float **latencyBufferOut = mAiffAllocateAudioBuffer(numChannels,
                          DiracFxMaxOutputBufferFramesRequired(time, pitch, latencyFrames));

      /* Read the first chunk from the file */
      mAiffReadData(infileName, latencyBufferIn, 0, latencyFrames, numChannels);

      /* The first block is processed manually to account for the latency */
      DiracFxProcessFloat(time, pitch, latencyBufferIn,
                                          latencyBufferOut, latencyFrames, diracFx);

      /* The first block is processed manually to account for the latency */
      /* but increase our read position */
      inputFramesProcessed += latencyFrames;

      /* ***************** MAIN PROCESSING LOOP STARTS HERE ******************* */
      for(;;) {

         /* read chunk at position inputFramesProcessed */
         mAiffReadData(infileName, audioIn, inputFramesProcessed, numFrames,
                                   numChannels);

         /* Call the process function with current time and pitch settings */
         /* Returns: the number of frames in audioOut */
         long ret = DiracFxProcessFloat(time, pitch, audioIn, audioOut,
                                                               numFrames, diracFx);

         /* Write data to the output file */
         mAiffWriteData(oufileName, audioOut, ret, numChannels);

         /* Increase our input position */
         inputFramesProcessed += numFrames;

         /* As soon as we've read enough frames we exit the main loop */
         if (inputFramesProcessed >= inputNumFrames + latencyFrames)
                 break;
      }
      /* ***************** END MAIN PROCESSING LOOP ******************* */

      /* ***************** CLEAN UP ******************* */
      /* Free processing buffers */
      mAiffDeallocateAudioBuffer(audioIn, numChannels);
      mAiffDeallocateAudioBuffer(audioOut, numChannels);
      mAiffDeallocateAudioBuffer(latencyBufferIn, numChannels);
      mAiffDeallocateAudioBuffer(latencyBufferOut, numChannels);

      /* Destroy DiracFx instance */
      DiracFxDestroy( diracFx );

      /* We're done! */
      printf("\nDone!\n");

#ifdef __APPLE__
      /* Open audio file via system call on the Mac */
      system("open out.aiff");
#endif
      return 0;
}
```

5

Chapter 5: Utility Functions

As of version 3, an increasing number of utility functions has been added to the Dirac core API. Following is a brief description.

Please note that calls that require an instance pointer work only on instances of Dirac, and neither on DiracRetune nor DiracFx!

5.1 Dirac Core Debugging Tools

```
void DiracPrintSettings(void *dirac);
```

Prints all relevant parameters for the instance pointed to by `dirac` to the console. This is handy for debugging, logging or sharing settings.

```
const char *DiracErrorToString(long error);
```

Returns a string containing a textual description of the Dirac error described by error. This can help make error messages more readable.

```
const char *DiracVersion(void);
```

Returns the current version number, build date and license type as string.

5.2 Measurement Tools

```
void DiracStartClock(void);
```

Starts and resets Dirac's internal stop watch. Note that there is only one stop watch implemented in Dirac at runtime.

```
long double DiracClockTimeSeconds(void);
```

Returns the number of seconds elapsed since the last call to `DiracStartClock()`

```
float DiracPeakCpuUsagePercent(void *dirac);
```

Returns the peak CPU usage in percent since the last call to `DiracPeakCpuUsagePercent()`. This also resets the CPU usage peak value so you can be sure that you are never missing spikes in CPU usage. This figure reflects the CPU usage for the entire Dirac call chain during processing, including the time spent in the callback (and all calls made until the callback returns).

CPU percentage is measured by comparing the time it takes to complete one call of `DiracProcess`() against the time it would take to play back the number of frames in numFrames at the current instance's sample rate.

5.3 Helpers

```
long double DiracValidateStretchFactor(long double factor);
```

Compares `factor` against the lower and upper boundaries of the current Dirac version and returns the corrected value. Dirac will call this function internally when setting its parameters, this call is intended to make sure the UI of your application will reflect the value that Dirac will be using as well. `factor` is the combined factor including both time and pitch scaling: factor = timeFactor * pitchFactor

6

Chapter 6: Getting in Touch

Contact us!

We would like to receive your feedback on this product, preferably through the contact form on our web site at http://www.dspdimension.com in the CONTACT area. Thank you for your interest in Dirac, we hope you like it and will enjoy the many features that it offers!

Stephan M. Bernsee
February 2012