



Free* Commercial Grade Cross-Platform Time and Pitch Manipulation Library for Polyphonic Audio

v2.2.5

© ClearScale.org, DIRAC, the DIRAC logo, object code and this user manual are © 2004-2010 Stephan M. Bernsee

No part of the manual and commercial DIRAC software version may be copied, sold, or otherwise reproduced for profit without the prior written consent of the author.

<http://www.dsdimension.com>

*) Please note that only the DIRAC LE version is available for free. The more powerful STUDIO and PRO versions are available for commercial licensing from the author. Please see the above web site for more information.

DIRAC LE License Agreement

PLEASE READ THIS LICENSE CAREFULLY BEFORE USING THE SOFTWARE. BY USING THE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE. This software is supplied to you by Stephan M. Bernsee in consideration of your agreement to the following terms, and your use or installation of this software constitutes acceptance of these terms. If you do not agree with these terms, please do not use, install or redistribute this software.

1. License.

a) The DIRAC LE object code, demonstration source code and other software accompanying this License, whether on disk, in read only memory, or on any other media (the 'Software'), the related documentation and example audio files are licensed to you by Stephan M. Bernsee.

b) Under the purpose of this agreement, Stephan M. Bernsee grants you a non-transferable, non-exclusive worldwide license (the "License") to

- i. use the Software for the purpose of changing length and pitch of audio signals in your own computer software product.
- ii. link and/or combine the Software with your own software project to produce an executable application (the "Integrated Product") that can be used by an end user.
- iii. copy and distribute, and have copied and distributed, to your customers portions of the Software embedded into or accompanying the Integrated Product, subject to the terms and conditions of this agreement.
- iv. grant end users non-exclusive licenses to use the Integrated Product, subject to the restrictions contained in this agreement.

c) Legal title to the Software, documentation and example files provided under this agreement shall remain in Stephan M. Bernsee as its sole property. Except as expressly stated in this notice, no other rights or licenses, express or implied, are granted by Stephan M. Bernsee herein, including but not limited to any patent rights that may be infringed by your derivative works or by other works in which the Software may be incorporated.

2. Restrictions.

a) The Software contains copyrighted material, trade secrets, and other proprietary material. In order to protect them, and except as permitted by applicable legislation, you may not decompile, reverse engineer, disassemble or otherwise reduce the Software to a human-perceivable form. You may not modify, rent, lease, loan or re-distribute the Software in whole or in part other than for the purpose detailed in §1b.

b) You agree to include the following copyright notice in all printed or electronic documentation accompanying the Integrated Product, as well as in all places within the Integrated Product's user interface where you are placing your own copyright notices and mentions the author(s) of the Integrated Product:

"DIRAC Time Stretch/Pitch Shift technology (c) 2005-2010 Stephan M. Bernsee"

In addition, you may indicate in the packaging, advertisements, and documentation for the Integrated Products that the Integrated Products contain the DIRAC technology.

3. Termination.

This License is effective until terminated. You may terminate this License at any time by destroying the Software, related documentation and example files and all copies thereof. This License will terminate immediately without notice from Stephan M. Bernsee if you fail to comply with any provision of this License. Upon termination you must destroy the Software, related documentation and example files and all copies thereof.

4. Disclaimer of Warranty on the Software.

You expressly acknowledge and agree that use of the Software and example files is at your sole risk. The Software, related documentation and example files are provided 'AS IS' and without warranty of any kind and Stephan M. Bernsee EXPRESSLY DISCLAIMS ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. STEPHAN M. BERNSEE DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE SOFTWARE AND THE EXAMPLE FILES WILL BE CORRECTED. FURTHERMORE, STEPHAN M. BERNSEE DOES NOT WARRANT OR MAKE ANY

REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE AND EXAMPLE FILES OR RELATED DOCUMENTATION IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY STEPHAN M. BERNSEE OR A STEPHAN M. BERNSEE AUTHORIZED REPRESENTATIVE SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY. WITHOUT LIMITING THE FOREGOING, STEPHAN M. BERNSEE DISCLAIMS ANY AND ALL EXPRESS OR IMPLIED WARRANTIES OF ANY KIND, AND YOU EXPRESSLY ASSUME ALL LIABILITIES AND RISKS, FOR USE OR OPERATION OF THE SOFTWARE, INCLUDING WITHOUT LIMITATION. SHOULD THE SOFTWARE PROVE DEFECTIVE, YOU (AND NOT STEPHAN M. BERNSEE OR A STEPHAN M. BERNSEE AUTHORIZED REPRESENTATIVE) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

5. Limitation of Liability.

UNDER NO CIRCUMSTANCES INCLUDING NEGLIGENCE, SHALL STEPHAN M. BERNSEE BE LIABLE FOR ANY INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES THAT RESULT FROM THE USE OR INABILITY TO USE THE SOFTWARE OR RELATED DOCUMENTATION, EVEN IF STEPHAN M. BERNSEE OR A STEPHAN M. BERNSEE AUTHORIZED REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU. In no event shall Stephan M. Bernsee's total liability to you for all damages, losses, and causes of action (whether in contract, tort (including negligence) or otherwise) exceed that portion of the amount paid by you which is fairly attributable to the Software and example files.

6. Controlling Law and Severability.

This License shall be governed by and construed in accordance with the laws of the Federal Republic of Germany. If for any reason a court of competent jurisdiction finds any provision of this License, or portion thereof, to be unenforceable, that provision of the License shall be enforced to the maximum extent permissible so as to effect the intent of the parties, and the remainder of this License shall continue in full force and effect.

7. Complete Agreement.

This License constitutes the entire agreement between the parties with respect to the use of the Software, the related documentation and fonts, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by Stephan M. Bernsee or a duly authorized representative of Stephan M. Bernsee.

Should you have any questions or comments concerning this license, please contact Stephan M. Bernsee at <http://www.dspdimension.com>

Copyright © 2005-2010 Stephan M. Bernsee, All Rights Reserved

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 DIRAC LE 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 multichannel- and multithreading-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 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^o

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, DIRAC is 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 might need to include the vecLib.Framework in your project). If you don't include these libraries 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 DIRAC Versions

There are 3 different incarnations of the desktop version of DIRAC available at this time. DIRAC LE is the free version of the library that provides the exact same audio quality as the STUDIO and PRO versions but has some minor usability restrictions. For example, DIRAC LE 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 DIRAC LE object per channel. DIRAC STUDIO offers phase locked stereo processing while the PRO version allows processing an arbitrary number of channels (5.1 surround and more) in a phase locked, mono-compatible manner.

^o Note that the commercial DIRAC STUDIO or PRO retail version is required in order to use this feature.

For the iPhone, only LE and PRO are available. The iPhone version can be licensed separately from the other platforms at a competitive price.

Below is a comparison chart of the 3 different DIRAC versions:

DIRAC2 Feature Comparison Chart

	DIRAC LE	DIRAC STUDIO	DIRAC PRO
Supported sample rates	44.1 + 48kHz	8 - 96 kHz	unlimited
Dynamic (time varying) time stretch and pitch shift	NO	NO	YES
Supported number of channels per DIRAC instance	1	1 + 2	unlimited
Pitch correction	pre-set equitempered scale only	pre-set equitempered scale only	any tuning table
Manual pitch	NO	NO	YES
Licensing fee	FREE	4,730 €	9,870 €

DIRAC2 iPhone Feature Comparison Chart

	DIRAC LE	DIRAC PRO	
Supported sample rates	44.1 + 48kHz	unlimited	
Dynamic (time varying) time stretch and pitch shift	NO	YES	
Supported number of channels per DIRAC instance	1	unlimited	
Pitch correction	pre-set equitempered scale only	any tuning table	
Manual pitch	NO	YES	Unlike with the desktop version, there is no STUDIO version for the iPhone
Licensing fee	FREE	1,000 €	

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

The basic DIRAC LE 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.

1.7 Can I license source code?

Our past experience shows that many 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

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, DIRAC STUDIO and PRO check for the availability of an AltiVec-enabled processor (“Velocity Engine”) to benefit from the additional vector processing unit on these systems. On these platforms you might need to add additional vector libraries to your project. Please see your compiler's manual for more information on using AltiVec support 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 DIRAC, we have created a processing framework that is entirely free from any delay – you don't have to worry about latency issues.

2.2 Quality, speed and localization properties

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:

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 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

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
kDiracQualityBetter	Very good quality mode but takes more CPU.
kDiracQualityBest	The highest quality mode. Note that this setting can be <i>very</i> slow.

2.3 Phase locked multi-channel processing vs. multiple channel processing^o

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,

^o Note that the commercial DIRAC STUDIO or PRO retail version is required in order to use this feature.

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.

With DIRAC LE, you can only process your material as separate channels with separate DIRAC objects.

2.4 Interleaved channel vs. individual channel format^o

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.

^o Note that this is only relevant for DIRAC STUDIO and PRO. DIRAC LE supports only one channel per instance.

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.

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.

3

API Description

The following section describes the DIRAC setup procedure and function calls.

3.1 DiracCreate

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

```
void *DiracCreateInterleaved(long lambda, long quality, long numChannels, float sampleRate, long (*readFromChannelsCallback)(float *data, long numFrames, 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^o

Defines the number of channels DIRAC PRO should process. There is no upper limit to this number (memory permitting). Note that DIRAC LE 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.

sampleRate^o

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 DIRAC LE will fail to create a DIRAC object if you pass anything other than a value

^o Note that this is only relevant for DIRAC STUDIO and PRO. DIRAC LE supports only one channel per instance.

^o DIRAC LE 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.

of 44100 and 48000. Please see the table in chapter 1.5 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.

The DIRAC 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.

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 should provide in `data[][]`

***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.*

3.1.1 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 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.

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

3.2 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^o) 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
<code>kDiracPropertyPitchFactor</code>	0.5 – 2.0	Set/get desired/current pitch shift factor (1.0=original, 0.5=octave down, 2.0=octave up)
<code>kDiracPropertyTimeFactor</code>	0.5 – 2.0	Set/get desired/current time stretch factor (1.0=normal, 0.5=double speed, 2.0=half speed)
<code>kDiracPropertyFormantFactor</code>	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
<code>kDiracPropertyCompactSupport</code>	0.0, 1.0	Set/get compact support for the time-frequency decomposition. 1.0 enables compact support (this is the default and need not be explicitly set), 0.0 disables compact support. 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.
<code>kDiracPropertyCacheGranularity</code>	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.
<code>kDiracPropertyDoPitchCorrection</code>	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. <i>This feature requires that the DIRAC lambda value is set to <code>kDiracLambda1</code> (musically monophonic instruments and voice)</i>
<code>KDiracPropertyPitchCorrectionBasicTuningHz</code>	400.0 – 500.0	Set/get desired/current reference tuning (in Hz) for the pitch correction. Default is 440 Hz. <i>This feature requires that the DIRAC lambda value is set to <code>kDiracLambda1</code> (musically monophonic instruments and</i>

^o Note that the commercial DIRAC STUDIO or PRO retail version is required in order to use this feature.

		<i>voice) and kDiracPropertyDoPitchCorrection is enabled.</i>
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). <i>This feature requires that the DIRAC lambda value is set to kDiracLambda1 (musically monophonic instruments and voice) and kDiracPropertyDoPitchCorrection is enabled.</i>
kDiracPropertyPitchCorrectionSlurSpeed	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. <i>This feature requires that the DIRAC lambda value is set to kDiracLambda1 (musically monophonic instruments and voice) and kDiracPropertyDoPitchCorrection is enabled.</i>

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 DIRAC LE 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 returns kDiracErrorFeatureNotSupported if the option you're trying to use is not available in the current context.

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

3.2.1 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.);
```

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.

3.2.2 Using DIRAC for 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^o) 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^o) processing. See section 3.2 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.

`kDiracPropertyPitchCorrectionSlurSpeed`

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.

^o Note that the commercial DIRAC STUDIO or PRO retail version is required in order to use this feature.

^o Note that the commercial DIRAC STUDIO or PRO retail version is required in order to use this feature.

3.3 DiracProcess

```
long DiracProcess(float **data, long numFrames, void *userData, void *dirac);  
long DiracProcessInterleaved(float *data, long numFrames, void *userData, 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 DIRAC LE 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.

*userData

A void* to a data structure or object that is passed on 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.

*dirac

void pointer to the allocated DIRAC object.

Please see the accompanying example project for more information

3.4 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` an `Dirac` object and frees all its associated memory.

`*dirac`

void pointer to the allocated DIRAC object.

3.5 DiracReset

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

`DiracReset` resets a DIRAC object optionally filling all its internal buffers with zeros (`clear == 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.

`clear`

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

`*dirac`

void pointer to the allocated DIRAC object.

Feedback

Contact us!

We would be happy to receive your feedback on this product through the contact form on our web site at <http://www.dspdimension.com> in the CONTACT area. Thank you for your interest in DIRAC and enjoy the many features that it offers!

Stephan M. Bernsee
April 2010