



RETUNE 1.0.4

by zplane.development

(c) 2018 zplane.development GmbH & Co. KG

February 9, 2018

Contents

1	reTune Documentation	2
1.1	Introduction	2
1.2	API Documentation	2
1.2.1	General	2
1.2.2	Naming Conventions	2
1.2.3	Instance Handling Functions	3
1.2.4	Parameter Setting Functions	3
1.2.5	Process Function	4
1.2.6	Usage Example	5
1.3	Support	6
2	Class Index	7
2.1	Class List	7
3	File Index	7
3.1	File List	7
4	Class Documentation	7
4.1	ReTunelf Class Reference	7
4.1.1	Detailed Description	8
4.1.2	Member Enumeration Documentation	8
4.1.3	Member Function Documentation	10
5	File Documentation	17
5.1	docugen.txt File Reference	17
5.1.1	Detailed Description	17
5.2	ReTunelf.h File Reference	17
5.3	ReTunelf.h	17

1 reTune Documentation

1.1 Introduction

reTune is zplane's real-time multi-pitch modification tool. It enables shifts of pitches belonging to the same pitch class to any other pitch class (e.g. a shift of all Cs to C#s). This enables a transformation of the input audio from any input key to any output key (e.g. from C major to C# minor). Since each pitch class can be mapped arbitrarily to any other pitch class, mappings not only include major and minor input and output keys, but any other scale such as church modes or pentatonic scales. reTune runs in real time and the pitch mapping can likewise be modified online during processing.

In addition to the mapping of individual pitches, reTune can correct each pitch by mapping notes to the nearest semitone thereby eliminating inaccurate intonation. - The API provides control over various parameters such as the sensitivity of the pitch detection, the detection of transients as well as the smoothing of the pitch contour.

Description of basic functionality

Operating system compliance

Structure of document

1.2 API Documentation

1.2.1 General

reTune is a real-time algorithm. That means that in each call to the process function, [ReTuneIf::ProcessData](#), the caller provides a number of input frames and reTune returns a number of output frames. Input and output of a single process call have a fixed latency. The caller can specify an output block size and given that the user provides the required number of input frames, each process call will write a full output block into the specified buffer. The required number of input frames needs to be retrieved by the function [ReTuneIf::GetFramesNeeded](#) before a process call and the correct number of frames should be passed to the process function to ensure the correct latency between input and output. Once all input frames have been provided, the remaining output blocks can be retrieved by the function [ReTuneIf::FlushBuffer](#).

Preferences for reTune can be set both before and during processing. This includes the mapping of semitones, the pitch correction, as well as all other processing parameters.

1.2.2 Naming Conventions

The following naming conventions are used throughout this manual:

A **frame** denotes the number of audio samples per channel, i.e. 512 stereo frames correspond to 1024 float values (samples).

A **pitch class** describes the name of a pitch independent of the octave it occurred in. In other words, notes with pitches at C0, C1, C2, etc. all belong to pitch class C.

Equal temperament is assumed for the analysis and the **tuning frequency** describes the

center frequency of the pitch A4 in this temperament system. If not set differently, a tuning frequency of 440 Hz is assumed. Specifying a different tuning frequency shifts the center frequencies of all pitches up or down.

1.2.3 Instance Handling Functions

- **static `Error_t ReTuneIf::CreateInstance` (`ReTuneIf*& pReTune`, `int iOutputBlockSize`, `float fSampleRate`, `int iNumOfChannels`)**

Creates a new instance of `reTune`. The handle to the new instance is returned in parameter `pReTune`. The output block size is given in frames in parameter `iOutputBlockSize`. `fSampleRate` denotes the input samplerate and `iNumOfChannels` contains the number of channels of the input audio.

If the function fails, the return value is different from `ReTuneIf::kNoError`.

The use of this function is required.

- **static `Error_t ReTuneIf::DestroyInstance` (`ReTuneIf*& pReTune`)**

Destroys the instance of `reTune` given in parameter `pReTune`.

If the function fails, the return value is different from `ReTuneIf::kNoError`.

The use of this function is required.

1.2.4 Parameter Setting Functions

- **`Error_t ReTuneIf::SetOffset`(`PitchClass ePitchClass`, `float fOffsetInSemitones`)**

Sets the offset for the pitch class given by `ePitchClass`. The offset is provided by `fOffsetInSemitones`. This function can be used to realize a mapping of an input pitch class to any output pitch class.

- **`Error_t ReTuneIf::SetEnablePitchCorrection`(`PitchClass ePitchClass`, `bool bEnable`)**

Enables/disables pitch correction for the pitch class given by `ePitchClass`. If `bEnable` is set to true, pitch correction is enabled, if set to false, it is disabled.

- **`Error_t ReTuneIf::SetPitchCorrection`(`float fPitchCorrectionFactor`)**

Sets the global pitch correction factor. Pitch correction is only applied to the enabled pitch classes (see `ReTuneIf::SetEnablePitchCorrection`). A pitch correction factor of 1 snaps all pitch contours to the center frequencies of the corresponding pitch classes. A pitch correction of 0 leaves the pitch contours unaltered. Any value in between 0 and 1 applies pitch correction proportionally.

- **`Error_t ReTuneIf::SetInputTuningFreq`(`float fTuningFreq`)**

Sets the tuning frequency of the input audio in Hz. The input tuning is used to calibrate the analysis of the input pitches and should hence match the input audio.

- **`Error_t ReTuneIf::SetOutputTuningFreq`(`float fTuningFreq`)**

Sets the tuning frequency of the output audio.

- **Error_t ReTuneIf::SetSensitivity(float fSensitivity)**

Sets how sensitive the algorithm should react to different f0 hypotheses. The parameter fSensitivity in dB specifies the level range below the most prominent f0 in which f0 hypotheses are taken into account. A sensitivity of 0 means that only the most prominent f0 hypothesis is taken into account, a value of -10 considers all f0 hypotheses up to 10dB below the most prominent f0. fSensitivity can be specified within the range [0dB..-40dB].

- **Error_t ReTuneIf::SetTransients(float fTransientVal)**

Sets the transient detection strength between (0..1). Low values mean that the algorithm doesn't react very sensitive to transients while high values mean that is very sensitive.

- **Error_t ReTuneIf::SetSmoothing(float fSmooth)**

Sets the smoothing time of pitch contours. A pitch shift might cause a gap in the pitch contour when an smooth transition is interrupted through a jump in pitch. The smoothing time specifies how quickly reTune will even out these gaps. - Short smoothing times allow quick jumps while longer smoothing times result in softer pitch transitions. This feature is particularly useful for vocals.

1.2.5 Process Function

- **int ReTuneIf::GetFramesNeeded()**

Returns the required number of input samples for the upcoming processing block. This function has to be called before each processing step to ensure correct input buffer sizes.

The method may be called with a new output buffer size as parameter. This will change the output buffer size until it is called with another buffer size again.

- **int ReTuneIf::ProcessData(const float* const* ppfInputBlock, int iNumOfInputFrames, float** ppfOutputBlock)**

Processes the input data and returns the output data at the specified latency. - The input as well as the output data are given as an array of pointers to the data. This means that, for example, ppfInputBlock[0] is a pointer to the float input data buffer of the first channel while ppfInputBlock[1] is the pointer to the float input data buffer of the second input channel. The range of the audio input data is +/-1.0f. The parameter iNumOfInputFrames describes the number of input frames. Please make sure that this parameter iNumOfInputFrames equals the value returned by the function [ReTuneIf::GetFramesNeeded\(\)](#) to ensure the requested output buffer size.

If the function fails, the return value is different from [ReTuneIf::kNoError](#). The use of this function is required.

- **int ReTuneIf::FlushBuffer(float** ppfOutputBlock)**

Gets all the remaining internal frames when no more input data is available and writes them into the buffer ppfOutputBlock. Returns the number of written samples.

The use of this function is optional.

- **int ReTuneIf::Reset()**

Sets all internal buffers to their initial state. The call of this function is needed before using the same instance of reTune for a different input signal. The use of this function is optional.

1.2.6 Usage Example

The command line example demonstrates how to use reTune to convert a song from major to minor or vice versa. The example application can be called by the following command

```
reTuneCl <inputFile> <outputFile> <rootPitch> <sourceScale> <targetScale>
```

The complete code can be found in the example source file reTuneClMain.cpp.

In the first step, we declare a pointer to the reTune instance and create an instance of the reTune class

```
ReTuneIf*          pInstanceHandle = 0;

// create class instance
ReTuneIf::CreateInstance (pInstanceHandle,
                          kOutBlockSize,
                          pCInputFile->GetSampleRate(),
                          pCInputFile->GetNumOfChannels());
```

We set the input-to-output pitch mapping by calling

```
GeneratePitchMapping (pInstanceHandle, commandLineParser.getParameter (
    "mapping"));
}
else
{
    GeneratePitchMapping (pInstanceHandle,
                          commandLineParser.getParameter ("key"),
                          commandLineParser.getParameter ("inputScale"),
                          commandLineParser.getParameter ("outputScale"));
```

with the source key and target key. Within this function, we call

```
pInstance->SetOffset ((ReTuneIf::PitchClass) sourceNote, semitoneDifference)
```

which allows us to define the mapping for each individual pitch class.

Next, we read chunks of data from our input file.

```
while (bReadNextFrame)
{
    int iInputSize = pInstanceHandle->GetFramesNeeded();

    iNumFramesRead = pCInputFile->Read (apfInputBuffer, iInputSize);
    if(iNumFramesRead<iInputSize)
    {
```

```

    for (ch=0; ch<pCInputFile->GetNumOfChannels(); ch++)
        memset (&apfInputBuffer[ch][iNumFramesRead], 0, (iInputSize-
iNumFramesRead)*sizeof(float));
        bReadNextFrame = false;
    }

```

Each chunk is processed

```

int iNumOfOutputFrames = pInstanceHandle->ProcessData (apfInputBuffer,
                                                    iInputSize,
                                                    apfOutputBuffer);

```

and the output is written to the output file.

```

if (iNumOfOutputFrames != kOutBlockSize)
{
    printf("\nerror!");
    return -1;
}

```

After all input frames have been processed, we obtain the remaining output files by calling

```

while ((iNumOfOutputFrames = pInstanceHandle->FlushBuffer (apfOutputBuffer))
> 0)
{
    if (iNumOfOutputFrames > 0 && commandLineParser.getParameter("output")
!= "" && pCOutputFile->IsFileOpen())
    {
        pCOutputFile->Write (apfOutputBuffer, iNumOfOutputFrames);
    }
}

```

Finally we destroy the [ReTuneIf](#) instance

```

ReTuneIf::DestroyInstance (pInstanceHandle);

```

This example demonstrates the basic use of the reTune API. Information about additional parameter settings can be found in [Parameter Setting Functions](#).

1.3 Support

Support for the source code is - within the limits of the agreement - available from:

zplane.development

grunewaldstr. 83

d-10823 berlin

germany

fon: +49.30.854 09 15.0

fax: +49.30.854 09 15.5

@: info@zplane.de

2 Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

[ReTuneIf](#) 7

3 File Index

3.1 File List

Here is a list of all files with brief descriptions:

[ReTuneIf.h](#) 17

4 Class Documentation

4.1 ReTuneIf Class Reference

```
#include <ReTuneIf.h>
```

Public Types

- enum [Error_t](#) { [kNoError](#), [kMemError](#), [kInvalidFunctionParamError](#), [kNotInitializedError](#), [kUnknownError](#), [kNumErrors](#) }
- enum [Version_t](#) { [kMajor](#), [kMinor](#), [kPatch](#), [kRevision](#), [kNumVersionInts](#) }
- enum [PitchClass](#) { [C](#), [CSharp](#), [D](#), [DSharp](#), [E](#), [F](#), [FSharp](#), [G](#), [GSharp](#), [A](#), [ASharp](#), [B](#), [kNumOfPitchClasses](#) }

Public Member Functions

- virtual int [ProcessData](#) (const float *const *ppfInputBlock, int iNumOfInputFrames, float **ppfOutputBlock)=0
- virtual int [FlushBuffer](#) (float **ppfOutputBlock)=0
- virtual int [Reset](#) ()=0
- virtual int [GetFramesNeeded](#) ()=0
- virtual int [GetFramesNeeded](#) (int iNewOutputBlockSize)=0
- virtual int [GetLatency](#) ()=0
- virtual [Error_t](#) [SetOffset](#) ([PitchClass](#) ePitchClass, float fOffsetInSemitones)=0
- virtual float [GetOffset](#) ([PitchClass](#) ePitchClass) const =0
- virtual [Error_t](#) [SetEnablePitchCorrection](#) ([PitchClass](#) ePitchClass, bool bEnable)=0
- virtual bool [GetEnablePitchCorrection](#) ([PitchClass](#) ePitchClass) const =0
- virtual [Error_t](#) [SetInputTuningFreq](#) (float fTuningFreq)=0

- virtual float [GetInputTuningFreq](#) () const =0
- virtual [Error_t SetOutputTuningFreq](#) (float fTuningFreq)=0
- virtual float [GetOutputTuningFreq](#) () const =0
- virtual [Error_t SetSensitivity](#) (float fSensitivity)=0
- virtual float [GetSensitivity](#) () const =0
- virtual [Error_t EnableShorterTransientMode](#) (bool bShouldBeEnabled)=0
- virtual [Error_t SetTransients](#) (float fTransientVal)=0
- virtual float [GetTransients](#) () const =0
- virtual [Error_t SetSmoothing](#) (float fSmooth)=0
- virtual float [GetSmoothing](#) () const =0
- virtual [Error_t SetPitchCorrection](#) (float fPitchCorrectionFactor)=0
- virtual float [GetPitchCorrection](#) () const =0

Static Public Member Functions

- static const int [GetVersion](#) (const [Version_t](#) eVersionIdx)
- static const char * [GetBuildDate](#) ()
- static [Error_t CreateInstance](#) ([ReTuneIf](#) *&pReTune, int iOutputBlockSize, float fSampleRate, int iNumOfChannels)
- static [Error_t DestroyInstance](#) ([ReTuneIf](#) *&pReTune)

4.1.1 Detailed Description

Definition at line 46 of file [ReTuneIf.h](#).

4.1.2 Member Enumeration Documentation

4.1.2.1 enum [ReTuneIf::Error_t](#)

Enumerator:

- kNoError*** no error occurred
- kMemError*** memory allocation failed
- kInvalidFunctionParamError*** one or more function parameters are not valid
- kNotInitializedError*** instance has not been initialized yet
- kUnknownError*** unknown error occurred
- kNumErrors***

Definition at line 50 of file [ReTuneIf.h](#).

```
{
    kNoError,
    kMemError,
    kInvalidFunctionParamError,
    kNotInitializedError,
    kUnknownError,
    kNumErrors
};
```

4.1.2.2 enum ReTuneIf::PitchClass

Enumerator:

C
CSharp
D
DSharp
E
F
FSharp
G
GSharp
A
ASharp
B
kNumOfPitchClasses

Definition at line 69 of file [ReTuneIf.h](#).

```
{  
    C,  
    CSharp,  
    D,  
    DSharp,  
    E,  
    F,  
    FSharp,  
    G,  
    GSharp,  
    A,  
    ASharp,  
    B,  
    kNumOfPitchClasses  
};
```

4.1.2.3 enum ReTuneIf::Version_t

Enumerator:

kMajor
kMinor
kPatch
kRevision
kNumVersionInts

Definition at line 60 of file [ReTuneIf.h](#).

```
{  
    kMajor,  
    kMinor,  
    kPatch,  
    kRevision,  
    kNumVersionInts  
};
```

4.1.3 Member Function Documentation

4.1.3.1 `static Error_t ReTuneIf::CreateInstance (ReTuneIf *& pReTune, int iOutputBlockSize, float fSampleRate, int iNumOfChannels)` [static]

creates an instance of zplane's reTune class

Parameters

<i>pReTune</i>	: returns a pointer to the class instance
<i>iOutput-BlockSize</i>	: desired max number of frames at the output, if not changed by GetFramesNeeded(.) this one is the default output block size, maximum is 1024 sample frames
<i>fSampleRate</i>	: input sample rate
<i>iNumOf-Channels</i>	: number of channels (currently limited to 2)

Returns

`ReTuneIf::Error_t` : returns an error or noError

4.1.3.2 `static Error_t ReTuneIf::DestroyInstance (ReTuneIf *& pReTune)` [static]

destroys an instance of the zplane's reTune class

Parameters

<i>pReTune</i>	: pointer to the instance to be destroyed
----------------	---

Returns

`ReTuneIf::Error_t` : returns an error or noError

4.1.3.3 `virtual Error_t ReTuneIf::EnableShorterTransientMode (bool bShouldBeEnabled)` [pure virtual]

Enable mode for input audio that contains short transients (e.g. piano).

Parameters

<i>bShouldBe-Enabled</i>	: enable flag
--------------------------	---------------

4.1.3.4 `virtual int ReTuneIf::FlushBuffer (float ** ppfOutputBlock)` [pure virtual]

gets the last frames in the internal buffer

Parameters

<i>ppfOutputBlock</i> ,:	double pointer to the output buffer of samples [channels][samples]
--------------------------	--

Returns

int : returns the number of frames returned

4.1.3.5 static const char* ReTuneIf::GetBuildDate () [static]

4.1.3.6 virtual bool ReTuneIf::GetEnablePitchCorrection (PitchClass *ePitchClass*) const [pure virtual]

returns if pitch correction is en/disbaled for each pitch class

Parameters

<i>ePitchClass</i>	: the pitch class
--------------------	-------------------

Returns

bool : returns if it is en-/disbaled

4.1.3.7 virtual int ReTuneIf::GetFramesNeeded () [pure virtual]

returns the sample frames needed for the next process call to complete

Returns

int : returns the number of sample frames

4.1.3.8 virtual int ReTuneIf::GetFramesNeeded (int *iNewOutputBlockSize*) [pure virtual]

returns the sample frames needed for the next process call to complete for a new output blocksize

Parameters

<i>iNewOutputBlockSize</i>	: the new output blocksize
----------------------------	----------------------------

Returns

int : returns the number of sample frames

4.1.3.9 virtual float ReTuneIf::GetInputTuningFreq () const [pure virtual]

returns the current input tuning frequency

Returns

float : returns the current input tuning frequency

4.1.3.10 virtual int ReTuneIf::GetLatency () [pure virtual]

returns the current latency of the algorithm

Returns

int : returns the latency in sample frames

4.1.3.11 virtual float ReTuneIf::GetOffset (PitchClass ePitchClass) const [pure virtual]

returns the currently chosen offset for each pitch class

Parameters

<i>ePitchClass</i> : the pitch class

Returns

float : returns the offset in semitones

4.1.3.12 virtual float ReTuneIf::GetOutputTuningFreq () const [pure virtual]

returns the current output tuning frequency

Returns

float : returns the current output tuning frequency

4.1.3.13 virtual float ReTuneIf::GetPitchCorrection () const [pure virtual]

returns the current pitch correction factor

Returns

float : returns the current pitch correction factor

4.1.3.14 virtual float ReTuneIf::GetSensitivity () const [pure virtual]

returns the current sensitivity

Returns

float : returns the current sensitivity

4.1.3.15 virtual float ReTuneIf::GetSmoothing () const [pure virtual]

returns the current smoothing speed

Returns

float : returns the current smoothing speed

4.1.3.16 virtual float ReTuneIf::GetTransients () const [pure virtual]

returns the current transient value (between 0..1.5)

Returns

float : returns the current transient value

4.1.3.17 static const int ReTuneIf::GetVersion (const Version_t eVersionIdx)
 [static]

4.1.3.18 virtual int ReTuneIf::ProcessData (const float *const * ppfInputBlock, int iNumOfInputFrames, float ** ppfOutputBlock) [pure virtual]

does the actual processing if the number of frames provided is as retrieved by [ReTuneIf::GetFramesNeeded\(\)](#) this function always returns the number of frames as specified when calling [ReTuneIf::CreateInstance\(..\)](#)

Parameters

<i>ppfInput-Block</i>	: double pointer to the input buffer of samples [channels][samples]
<i>iNumOf-InputFrames</i>	: the number of input frames
<i>ppfOutput-Block</i>	: double pointer to the output buffer of samples [channels][samples]

Returns

int : returns the number of frames returned

4.1.3.19 virtual int ReTuneIf::Reset () [pure virtual]

clears the internal buffers of the reTune algorithm. you call this to avoid the remaining samples in the process buffer being audible when you e.g. stop playback and start it again at an other time position. other parameters (pitch offsets, sensitivity, etc.) are not reset.

Returns

int : currently returns 0

4.1.3.20 virtual Error_t ReTuneIf::SetEnablePitchCorrection (PitchClass ePitchClass, bool bEnable) [pure virtual]

en-/disables the pitch correction for each pitch class

Parameters

<i>ePitchClass</i>	: the pitch class
<i>bEnable</i>	: enable or disable it

Returns

ReTuneIf::Error_t : returns an error or noError

4.1.3.21 virtual Error_t ReTuneIf::SetInputTuningFreq (float fTuningFreq) [pure virtual]

sets the tuning frequency of the input audio, this is important to do a proper input frequency to pitch mapping

Parameters

<i>fTuningFreq</i>	: the input tuning frequency
--------------------	------------------------------

Returns

ReTuneIf::Error_t : returns an error or noError

4.1.3.22 virtual Error_t ReTuneIf::SetOffset (PitchClass ePitchClass, float fOffsetInSemitones) [pure virtual]

sets the pitch offset for each pitch class in semitones

Parameters

<i>ePitchClass</i>	: the pitch class
<i>fOffsetInSemitones</i>	: the offset in semitones

Returns

[ReTuneIf::Error_t](#) : returns an error or noError

4.1.3.23 `virtual Error_t ReTuneIf::SetOutputTuningFreq (float fTuningFreq)`
`[pure virtual]`

sets the tuning frequency of the output audio

Parameters

<i>fTuningFreq</i>	: the output tuning frequency
--------------------	-------------------------------

Returns

[ReTuneIf::Error_t](#) : returns an error or noError

4.1.3.24 `virtual Error_t ReTuneIf::SetPitchCorrection (float`
`fPitchCorrectionFactor) [pure virtual]`

sets the pitch correction for the enabled pitch classes. 0 means that no pitch correction is performed while 1 means that each pitch in an enabled pitch class is moved to the center pitch of the corresponding pitch class.

Parameters

<i>fPitchCorrectionFactor</i>	: pitch correction factor between (0..1)
-------------------------------	--

Returns

[ReTuneIf::Error_t](#) : returns an error or noError

4.1.3.25 `virtual Error_t ReTuneIf::SetSensitivity (float fSensitivity)`
`[pure virtual]`

sets how sensitive the algorithm should react to different f0 hypotheses (0..1). 0 means that it will be only take the strongest f0 hypothesis while higher values mean that more f0 hypotheses are taken as f0s - this may lead to too many f0s, usually 0.25 is a good value

default is 0.25

Parameters

<i>fSensitivity</i>	: the sensitivity (0..1)
---------------------	--------------------------

Returns

[ReTuneIf::Error_t](#) : returns an error or noError

4.1.3.26 `virtual Error_t ReTuneIf::SetSmoothing (float fSmooth) [pure virtual]`

sets the smoothing speed for transitions between shifted notes. If a note is shifted the transition may be smoothed in order to avoid jumps - this is especially useful for vocals. default is 46ms

Parameters

<i>fSmooth</i>	: smoothing speed in ms (0ms..300ms)
----------------	--------------------------------------

Returns

[ReTuneIf::Error_t](#) : returns an error or noError

4.1.3.27 `virtual Error_t ReTuneIf::SetTransients (float fTransientVal) [pure virtual]`

sets the transient behavior strength between (0..1.5). Values below 1.0 mean that the algorithm should attenuate transients while values above 1.0 mean that transients should be amplified. default is 1.0

For input audio that contains short transients (e.g. piano), set the shorter transients mode flag.

Parameters

<i>fTransient-Val</i>	: a value between 0..1.5
<i>bEnable-Shorter-Transients-Mode</i>	: enable better preservation for short transients.

Returns

[ReTuneIf::Error_t](#) : returns an error or noError

The documentation for this class was generated from the following file:

- [ReTuneIf.h](#)

5 File Documentation

5.1 docugen.txt File Reference

5.1.1 Detailed Description

source documentation main file

Definition in file [docugen.txt](#).

5.2 ReTunelf.h File Reference

Classes

- class [ReTuneIf](#)

5.3 ReTunelf.h

```
00001
00002 //      /*! \file ReTune.h: \brief interface of the reTune class. */
00003 //
00004 // Copyright (c) 2000-2011
00005 //      zplane.development GmbH & Co. KG
00006 //
00007 // CONFIDENTIALITY:
00008 //
00009 //      This file is the property of zplane.development.
00010 //      It contains information that is regarded as privilege
00011 //      and confidential by zplane.development.
00012 //      It may not be publicly disclosed or communicated in any way without
00013 //      prior written authorization by zplane.development.
00014 //      It cannot be copied, used, or modified without obtaining
00015 //      an authorization from zplane.development.
00016 //      If such an authorization is provided, any modified version or
00017 //      copy of the software has to contain this header.
00018 //
00019 // WARRANTIES:
00020 //      This software is provided as << is >>, zplane.development
00021 //      makes no warranty express or implied with respect to this software,
00022 //      its fitness for a particular purpose or its merchantability.
00023 //      In no case, shall zplane.development be liable for any
00024 //      incidental or consequential damages, including but not limited
00025 //      to lost profits.
00026 //
00027 //      zplane.development shall be under no obligation or liability in respect
00028 //      of
00029 //      any infringement of statutory monopoly or intellectual property
00030 //      rights of third parties by the use of elements of such software
00031 //      and User shall in any case be entirely responsible for the use
00032 //      to which he puts such elements.
00033 //
00034 // CVS INFORMATION
00035 //
00036 // $RCSfile: reTune.h,v $
00037 // $Author: lerch $
00038 // $Date: 2009/02/18 14:38:14 $
00039 //
00040 //
00042
00043 #if !defined(__Inc_ReTuneIf__)
00044 #define __Inc_ReTuneIf__
00045
00046 class ReTuneIf
```

```
00047 {
00048 public:
00049
00050     enum Error_t
00051     {
00052         kNoError,
00053         kMemError,
00054         kInvalidFunctionParamError,
00055         kNotInitializedError,
00056         kUnknownError,
00057         kNumErrors
00058     };
00059
00060     enum Version_t
00061     {
00062         kMajor,
00063         kMinor,
00064         kPatch,
00065         kRevision,
00066         kNumVersionInts
00067     };
00068
00069     enum PitchClass
00070     {
00071         C,
00072         CSharp,
00073         D,
00074         DSharp,
00075         E,
00076         F,
00077         FSharp,
00078         G,
00079         GSharp,
00080         A,
00081         ASharp,
00082         B,
00083         kNumOfPitchClasses
00084     };
00085
00086
00087     static const int  GetVersion (const Version_t eVersionIdx);
00088     static const char* GetBuildDate();
00089
00090
00091     static Error_t CreateInstance (ReTuneIf*& pReTune, int iOutputBlockSize,
00101 float fSampleRate, int iNumOfChannels);
00102
00110     static Error_t DestroyInstance (ReTuneIf*& pReTune);
00111
00122     virtual int ProcessData (const float* const* ppfInputBlock, int
00123 iNumOfInputFrames, float** ppfOutputBlock) = 0;
00124
00132     virtual int FlushBuffer(float** ppfOutputBlock) = 0;
00133
00141     virtual int Reset() = 0;
00142
00148     virtual int GetFramesNeeded() = 0;
00149
00157     virtual int GetFramesNeeded(int iNewOutputBlockSize) = 0;
00158
00164     virtual int GetLatency() = 0;
00165
00174     virtual Error_t SetOffset (PitchClass ePitchClass, float fOffsetInSemitones
00175 ) = 0;
00183     virtual float  GetOffset (PitchClass ePitchClass) const = 0;
00184
00193     virtual Error_t SetEnablePitchCorrection (PitchClass ePitchClass, bool
00194 bEnable) = 0;
00202     virtual bool  GetEnablePitchCorrection (PitchClass ePitchClass) const = 0
00203 ;
00203
```

```
00211     virtual Error_t SetInputTuningFreq (float fTuningFreq) = 0;
00212
00219     virtual float  GetInputTuningFreq() const = 0;
00220
00228     virtual Error_t SetOutputTuningFreq (float fTuningFreq) = 0;
00229
00235     virtual float  GetOutputTuningFreq() const = 0;
00236
00247     virtual Error_t SetSensitivity (float fSensitivity) = 0;
00248
00255     virtual float  GetSensitivity() const = 0;
00256
00262     virtual Error_t EnableShorterTransientMode (bool bShouldBeEnabled) = 0;
00263
00275     virtual Error_t SetTransients (float fTransientVal) = 0;
00276
00283     virtual float  GetTransients() const = 0;
00284
00294     virtual Error_t SetSmoothing (float fSmooth) = 0;
00295
00302     virtual float  GetSmoothing() const = 0;
00303
00311     virtual Error_t SetPitchCorrection(float fPitchCorrectionFactor) = 0;
00312
00319     virtual float  GetPitchCorrection() const = 0;
00320
00321 };
00322
00323 #endif // #if !defined(__Inc_ReTuneIf__)
00324
00325
00326
```