



espace parametric realtime convolution SDK
documentation

Alexander Lerch

(c) 2004 by zplane.development

September 21, 2004

Contents

1	espace parametric convolution SDK documentation	1
1.1	Introduction	1
1.1.1	Features on a view	1
1.2	API Documentation	2
1.2.1	Memory Allocation	2
1.2.2	Naming Conventions	2
1.2.3	Required Functions	2
1.2.4	Parameter Control Functions	3
1.2.5	Usage example	5
1.2.6	Parameter Description	7
1.3	MSVC Workspace Convolver.dsw	9
1.3.1	File Structure	9
1.3.2	Project Structure	10
1.3.3	Project Configurations	10
1.4	Coding Style minimal overview	10
1.5	Licensing Issues	11
1.6	Graph Element descriptions	11
1.7	Support	12
2	espace realtime convolution SDK Class Index	12
2.1	espace realtime convolution SDK Class List	12
3	espace realtime convolution SDK File Index	12
3.1	espace realtime convolution SDK File List	12
4	espace realtime convolution SDK Class Documentation	13
4.1	_EnvelopePoint_ Struct Reference	13
4.1.1	Detailed Description	13
4.2	espaceIf Class Reference	13
4.2.1	Detailed Description	13
4.2.2	Member Function Documentation	14
5	espace realtime convolution SDK File Documentation	19
5.1	docugen.txt File Reference	19

5.1.1	Detailed Description	19
5.2	espaceAPI.h File Reference	20
5.2.1	Detailed Description	20
5.2.2	Typedef Documentation	20
5.2.3	Enumeration Type Documentation	21

1 espace parametric convolution SDK documentation

1.1 Introduction

Using the espace SDK, it is possible to process convolution in realtime and for multichannel files. Besides the efficient convolution, espace allows to set parameters for the impulse response just like with conventional artificial reverbs - with the difference that no artificial reverb is created but only the impulse response itself is modified. This combines the high quality of convolution reverbs with the flexibility of artificial reverb and makes espace a powerful and intuitive reverb in nearly all areas of application. An overview of the features is given below.

The SDK comes with a C++-style API.

espace is available for all common x86 and PPC operating systems. Apart from the delivered libraries, no additional libraries are needed. The SDK is delivered with the libraries, the corresponding header files, this documentation and a small example application.

The first part of this document explains the API of the SDK. Then, a small usage example explaining the basic usage of the SDK is given. After some additional notes, some automatically generated documentation follows.

1.1.1 Features on a view

- high performance due to assembler optimizations
- no restrictions on impulse response regarding sample rate, number of channels, length
- automatic analysis of impulse response to achieve a correct crosspoint between early reflections (ER) and late reverberation (LR)
- automatic sample rate conversion if the sample rate of the impulse response does not equal the processing sample rate
- modification of complete impulse response with the following parameters:
 - Pre-Delay
 - N-point envelope
- modification of ER and LR parts of the impulse response individually with the following parameters:

- Length without changing pitch
- Pitch without changing length
- up to ten filters with different settings/characteristics (Lowpass, Highpass, Bandpass, Low-shelving, High-shelving, Peak)
- user adjustable cross-point and cross-fade time between ER and LR part of impulse response
- automatic volume compensation to avoid clipping
- setting of processing block size (latency), wetness, input downmixing to mono
- optional reverse impulse response processing
- possibility to calculate parameter changes in background while processing convolution

1.2 API Documentation

The interface of the espace parametric convolution SDK is given in the file [espace-API.h](#). There are no additional headers required.

1.2.1 Memory Allocation

The espace SDK does not allocate buffers handled by the calling application. The input and output buffers have to be allocated by the calling application. Audio buffers are allocated as double arrays of [channels][SamplesPerChannel].

1.2.2 Naming Conventions

Before using the SDK, an **instance** has to be created. After successful creation, the other API functions can be called.

When talking about frames, the number of audio samples per channel is meant. For mono signals, the number of frames equals the number of samples. For stereo signals, the number of samples is twice the number of frames. So, in general, the number of samples equals the number of frames times the number of channels. If the sample size is 16bit, one sample has a memory usage of 2 byte.

1.2.3 Required Functions

The following functions have to be called when using the espace library:

- **zERROR [espaceIf::CreateInstance](#) ([espaceIf*](#)& [pCespace](#))**

This function creates a new instance of espace and has to be called before anything can happen. The parameter `pCespace` is the handle to the new instance and is set in the function.

The function returns 0 when no error occurred.

- **zERROR [espaceIf::DestroyInstance](#) ([espaceIf*& pCespace](#))**

This function destroys an instance of `espace` and has to be called after everything has been done to free the allocated memory etc. The parameter `pCespace` is the handle to the instance to be destroyed and is set in the function to 0.

- **zERROR [espaceIf::Initialize](#) ([int iSampleRate](#), [int iNumOfChannels](#), [int iBlockSize](#))**

This function initializes an instance of `espace` and has to be called after the instance has been created. The parameter `iSampleRate` is the processing sample rate in Hz, the parameter `iNumOfChannels` is the number of processing channels, and the parameter `iBlockSize` is the processing frame size that has to be a power of two.

The function returns 0 when no error occurred.

- **zERROR [espaceIf::Process](#) ([const float **ppfInputData](#), [float **ppfOutputData](#), [int iNumOfFrames](#))**

This function does the actual convolution. The pointer `ppfInputData` contains in input data in an array `[channels][samples]`, the output data is written to the buffer pointed to in parameter `ppfOutputData` of the same array format. Note the the parameter `ppfOutputData` may equal the parameter `ppfInputData` for inplace processing. The parameter `iNumOfFrames` is the number of frames to process. Please note that before this function can be called, the impulse response has to be set via the function [espaceIf::SetImpulseResponse](#).

The function returns 0 when no error occurred.

- **zERROR [espaceIf::SetImpulseResponse](#) ([const float **ppfImpulseResponse](#), [int iNumOfIRFrames](#), [int iNumOfIRChannels](#), [int iIRSampleRate](#))**

This function is used to set the impulse response. The impulse response data is in an array pointed to by parameter `ppfImpulseResponse` in the format `[channels][samples]`, the parameter `iNumOfIRFrames` is the number of frames of the impulse response, the parameters `iNumOfIRChannels` and `iIRSampleRate` contain the number of impulse response channels resp. the sample rate of the impulse response. The number of channels must equal the number of process channels if this number is not 1. If the sample rate of the impulse response does not equal the processing sample rate, it will be automatically converted to the processing sample rate.

The function returns 0 when no error occurred.

1.2.4 Parameter Control Functions

The following functions are to set or get the individual parameter values. Most parameters can be controlled via the function [espaceIf::SetParameter](#), however, some parameters are controlled via extra functions.

- **zERROR [espaceIf::SetParameter](#) ([int iIndex](#), [float fValue](#))**

This function allows to set nearly all available parameter (see section [Parameter Description](#)). The parameter `iIndex` gives the parameter index, and the parameter `fValue` the value of the parameter.

The function returns 0 when no error occurred.

- **float `espaceIf::GetParameter` (int `iIndex`)**

This function is to get the available parameter values (see section [Parameter Description](#)). The parameter `iIndex` gives the parameter index, its value is returned.

- **zERROR `espaceIf::SetStretchnPitch` (float `fStretchFactor`, float `fPitchFactor`, `_IRPart_ ePart`)**

This function can be used to set the stretch and/or pitch factor for either the ER or the LR part of the impulse response. The parameter `fStretchFactor` is the stretch factor in percent (range 0.5...1.5), representing the resulting length of the impulse response, the parameter `fPitchFactor` is the pitch shifting in percent with the same range, and the parameter `ePart` is to define if these changes should take effect on the ER or the LR part of the impulse response.

The function returns 0 when no error occurred.

- **zERROR `espaceIf::CreateFilter` (void `**phFilterHandle`, `_FilterTypes_ eType`, `_IRPart_ ePart`)**

This function is to create a new filter to be applied to either the ER or the LR part of the impulse response. The parameter `phFilterHandle` is the handle to the new filter (to be written), the parameter `eType` defines the filter type (see [_FilterTypes_](#)), and the parameter `ePart` is to define if these changes should take effect on the ER or the LR part of the impulse response.

The function returns 0 when no error occurred.

- **zERROR `espaceIf::DestroyFilter` (void `**phFilterHandle`)**

This function is to destroy a previously created filter. The parameter `phFilterHandle` is the handle to the filter.

The function returns 0 when no error occurred.

- **zERROR `espaceIf::SetFilterParameter` (void `*phFilterHandle`, float `fFilterFreq`, float `fFilterGain = 0`, float `fFilterQ = 1`)**

This function sets the parameters of a previously created filter. The parameter `phFilterHandle` is the handle to the filter and the parameter `fFilterFreq` specifies, dependent on the selected filter type, the mid or cut-off frequency of the filter in Hz. The parameters `fFilterGain` and `fFilterQ` have to be set only in the case that the selected filter type requires them; they correspond to the gain in dB and the Q of the filter.

The function returns 0 when no error occurred.

- **zERROR `espaceIf::GetFilterParameter` (void `*phFilterHandle`, float `*pfFilterFreq`, float `*pfFilterGain = 0`, float `*pfFilterQ = 0`)**

This function is to return the parameters of a previously created filter. For the parameter description, see function `espaceIf::SetfilterParameter`, except that in this case all parameters are pointer that are written by this function.

The function returns 0 when no error occurred.

- **zERROR** `espaceIf::SetEnvelope` (`_EnvelopePoint_ **ppstEnvelopePoints`, `int iNumOfEnvelopePoints`)

This function allows to set an envelope for the impulse response. The parameter `ppstEnvelopePoints` contains pointers to structures where the envelope points and their values are defined (see also `_EnvelopePoint_`), and the parameter `iNumOfEnvelopePoints` specifies the number of envelope points. The number of envelope points must exceed a value of two (beginning and end). The envelope itself is a piecewise linear function that goes through all specified points.

The function returns 0 when no error occurred.

- **int** `espaceIf::GetEnvelope` (`_EnvelopePoint_ **ppstEnvelopePoints`)

This function returns the specified envelope for the impulse response. The parameter `ppstEnvelopePoints` is exactly the same as described in the function `espaceIf::SetEnvelope`; its memory has to be allocated by the user beforehand. The function returns the number of envelope points; if the argument `ppstEnvelopePoints` is 0, only the number of envelope points is returned.

- **zERROR** `espaceIf::ApplyParameterChanges` (`PFUNCallback pCallbackWhenReady = 0`, `void *pUserData = 0`)

This function has to be called after every parameter change to let the changes take effect. It may be called in a separate thread to allow background calculation of the new impulse response while the old impulse response is processed normally, i.e. `espaceIf::Process` can be called while this function is processing. In the first call of `espaceIf::Process` after exiting this function, the impulse response is switched from the previous to the new one. To be noticed when the background process has finished (optionally), it is possible to optionally specify a function pointer of type `PFUNCallback` in the parameter `pCallbackWhenReady` and some user defined pointer `pUserData`. then, after the background process has finished, this function is called with the specified user data pointer.

The function returns 0 when no error occurred.

1.2.4.1 Other Functions

- **zERROR** `espaceIf::Reset` ()

Upon the call of this function, all internal buffers are flushed to 0.

The function returns 0 when no error occurred.

1.2.5 Usage example

The complete code can be found in the example source file `ConvolverTestCLMain.cpp`. The example application uses the open source library `libSndFile` for audio file format parsing.

In the first step, a pointer to the needed `espaceIf` instance has to be declared. This can be done with the following code snippet:

```
espaceIf          *pConvolveInstance = 0;           // instance handle for espace
```

First, a new instance has to be created:

```
// create a new instance of espace and do initialization
if (espaceIf::CreateInstance (pConvolveInstance) != 0)
{
    fprintf(stdout, "Instance creation failed!\n");
    sf_close (pOutputFile);
    sf_close (pFIRFile);
    sf_close (pInputFile);
    return -1;
}
```

which has to be initialized with sample rate, number of channels, and the required blocksize

```
pConvolveInstance->Initialize ( sfInputInfo.samplerate,
                               sfInputInfo.channels,
                               _BLOCKSIZE);
```

Now, the impulse response can be read from file or memory, sorted to the correct input format and handed over to the SDK:

```
// read impulse response data and sort the data
sf_readf_float (pFIRFile, pfIRDataInterleaved, iLengthOfIR);
h = 0;
for ( i = 0; i < iLengthOfIR; i++)
    for(j = 0; j < sfIRInfo.channels; j++)
        pfIRData[j][i] = pfIRDataInterleaved[h++];

// set the impulse response
pConvolveInstance->SetImpulseResponse ( (const float**)pfIRData,
                                       iLengthOfIR,
                                       sfIRInfo.channels,
                                       sfIRInfo.samplerate);
```

At this stage, parameters can be set, as can be seen in the following examples. In the first example, an envelope with only three points is defined and set (actually, this envelope consists only of a flat line and does nothing at all)

```
// generate and set - as an example - three envelope points
for (i = 0; i < _NUM_ENVELOPE_POINTS; i++)
```

Setting the time stretching and pitch shifting factors for the early reflections can be done with the following code (which in this special case results in no alteration of the impulse response again):

```
// set time-stretching factors
pConvolveInstance->SetStretchnPitch ( 1.0F,
                                     1.0F,
                                     ER);
```

To create a new filter and set its parameters (in this for the early reflections part), the code would look like

```
// create and set one filter
pConvolveInstance->CreateFilter (      &pvERFilter,
                                       FilterTypePeak,
                                       ER);
pConvolveInstance->SetFilterParameter ( pvERFilter,
                                       7000,
                                       6,
                                       1);
```

Other parameters can be set via the `::SetParameter` function, here is an example for the `PreDelay`

```
pConvolveInstance->SetParameter (kParamPreDelay, 30.0F);
```

After successfully setting the parameters, the function `::ApplyParameterChanges` has to be called to let the changes take effect. This function can be called in a separate thread to allow processing while calculating the new impulse response in background.

```
// now apply the parameter changes to the impulse response
pConvolveInstance->ApplyParameterChanges (CallbackFunction);
```

To request either information about the impulse response analysis or the current parameter state, the function `::GetParameter` can be called

```
fprintf(      stderr,
```

To process a new block of audio data, the process function has to be called

```
// begin of the processing loop
while(bReadNextFrame)
{
    // read samples from input file
    iNumFramesRead = (int)(sf_readf_float( pFInputFile,
                                           aFloatData,
                                           _BLOCKSIZE));
```

After the processing, the created filters and the instance of espace realtime convolution have to be destroyed

```
// destroy used filters
pConvolveInstance->DestroyFilter (&pvERFilter);
```

The above code snippets demonstrated the basic functionality of the espace library. Most of the additional functions can be used similar to the given code examples. The exact functionality of the functions is described above.

1.2.6 Parameter Description

Despite some special parameters like time stretching, filtering and envelope, the parameters can be set via the function `::SetParameter` which first parameter is the index of the

parameter itself. The parameter indices are listed in the enum `_Parameter_Indices_` in the header file `espaceAPI.h`. Note that there are two distinct types of parameters, those that can be used to set some values and those that can only be requested because they are analysis results.

All parameter values are in float format; if the parameter is only on/off, then the allowed values are 0.0 (false/off) and 1.0 (true/on)

1.2.6.1 Changeable Parameters (alphabetical)

- `kParamAutoVolumeScaleEnabled`: enable/disable automatic volume scale to avoid clipping (allowed values 0 (off) and 1 (on), default: 1); please note that the scaling is dependent on the input signal and a slight correction of the output gain may be necessary nevertheless
- `kParamBlockSize`: set/get the processing block size in frames, has to be a power of two
- `kParamBypass`: enable/disable bypass (allowed values 0 (off) and 1 (on), default: 0)
- `kParamInputMode`: enable/disable downmixing of input to mono (allowed values 0 (no downmix) or 1 (downmix), default: 0)
- `kParamPreDelay`: set/get Pre-Delay (delay of impulse response) in ms (range 0...300, default: 0)
- `kParamPreserveLength`: enable/disable the truncation of "unnecessary" frames at the begin and end of the impulse response (allowed values 0 (off) and 1 (on), default: 1)
- `kParamResLPCutOff`: set/get cut-off frequency of resonator low pass to be applied to the output signal in Hz (default: half sample rate)
- `kParamResLPModDepth`: set/get second cut-off frequency of resonator low pass to be applied to the output signal in Hz (default: half sample rate), the cut-off frequency is modulated between `kParamResLPCutOff` and `kParamResLPModDepth`
- `kParamResLPModFreq`: set/get modulation frequency of resonator low pass to be applied to the output signal in Hz (range 0...20, default: 0)
- `kParamResLPResonance`: set/get resonance of resonator low pass to be applied to the output signal (range 0 (no resonance)...4 (resonance), default: 0)
- `kParamReverseIR`: enable/disable to swap the whole impulse response from end to start (allowed values 0 (off) and 1 (on), default: 0)
- `kParamXFadeLength`: set/get crossfade length between ER and LR in ms (default: 50ms)
- `kParamXPoint`: set/get crosspoint between ER and LR in percent (range 0...1, 0 means start, 1 means end, default: impulse response dependent)
- `kParamWetness`: set/get wetness (dry/wet) in percent (range 0...1, 0 means dry, 1 means wet, default: 1)

1.2.6.2 Non-Changeable Parameters (alphabetical)

- **kParamProcessLatency**: get processing latency (equals block size) in s (see **kParamBlockSize**)
- **kParamLengthOfOrigIR**: get length of loaded impulse response in s (cannot be set)
- **kParamLengthOfCurrentIR**: get length of current processed (modified) impulse response in s (can be set only implicitly via stretch factors)
- **kParamReverberationTime**: get measured reverberation time in s (cannot be set)
- **kParamNumOfTailSamples** : get length of tail in frames (i.e. how many zeros have to be fed to the library to get the complete reverberation tail)
- **kParamStretchER**: get the stretch factor of early reflections in percent (can be set via `::SetStretchnPitch`)
- **kParamPitchER**: get the pitch factor of early reflections in percent (can be set via `::SetStretchnPitch`)
- **kParamStretchLR**: get the stretch factor of late reverberation in percent (can be set via `::SetStretchnPitch`)
- **kParamPitchLR**: get the pitch factor of late reverberation in percent (can be set via `::SetStretchnPitch`)

1.3 MSVC Workspace Convolver.dsw

1.3.1 File Structure

1.3.1.1 Documentation This documentation and all other documentation can be found in the directory `./doc`.

1.3.1.2 Project Files The MS VisualC++-Workspace (`.dsw`) and all Projectfiles (`.dsp`) can be found in the directory `./build` and its subfolders, where the subfolders names correspond to the project names.

1.3.1.3 Source Files All source files are in the directory `./src` and its subfolders, where the subfolder names equally correspond to the project names.

1.3.1.4 Include Files If include files are project intern, they are in the source directory `./src` of the project itself. If include files are to be included by other projects they can be found in `./src/include`. If a SDK-like library/DLL is built for which a header is needed, such a header can be found in `./inc`.

1.3.1.5 Resource Files The resource files can be found in the subdirectory `/res` of the corresponding build-directory.

1.3.1.6 Library Files The directory `./lib` is for used and built libraries.

1.3.1.7 Binary Files The final executable as well as the distributable Dynamic Link Libraries can be found in the directory `./bin`. In debug-builds, the binary files are in the subfolder `/Debug`.

1.3.1.8 Temporary Files The directory `./tmp` is for all temporary files while building the projects. In debug-builds, the temporary files can be found in the subfolder `/Debug`.

1.3.2 Project Structure

The project structure is as following:

- **ConvolverTestCL**: the command line example application: The project output is an executable binary (EXE).
- **espaceVST**: VST plugin with rudimentary GUI as coding example: The project output is a Dynamic Link Library (DLL).

Both projects use the open source library libSndFile (LGPL) for audio file IO.

1.3.3 Project Configurations

For all projects included in the workspace, the default configurations Win32 Release and Win32 Debug are available.

1.4 Coding Style minimal overview

Variable names have a preceding letter indicating their types:

unsigned:	u
pointer:	p
array:	a
class:	C
bool:	b
char:	c
short (int16):	s
int (int32):	i
__int64:	l
float (float32):	f
double (float64):	d
char:	c

For example, a pointer to a buffer of unsigned ints will be named `puiBufferName`.

1.5 Licensing Issues

Please note that there might be intellectual property rights on the algorithms used. zplane cannot be held responsible for any possible infringements.

1.6 Graph Element descriptions

This is an excerpt from the doxygen documentation:

The elements in the class diagrams in HTML and RTF have the following meaning:

- *A yellow box indicates a class. A box can have a little marker in the lower right corner to indicate that the class contains base classes that are hidden. For the class diagrams the maximum tree width is currently 8 elements. If a tree is wider some nodes will be hidden. If the box is filled with a dashed pattern the inheritance relation is virtual.*
- *A white box indicates that the documentation of the class is currently shown.*
- *A grey box indicates an undocumented class.*
- *A solid dark blue arrow indicates public inheritance.*
- *A dashed dark green arrow indicates protected inheritance.*
- *A dotted dark green arrow indicates private inheritance.*

The elements in the class diagram in PDF have the following meaning:

- *A white box indicates a class. A marker in the lower right corner of the box indicates that the class has base classes that are hidden. If the box has a dashed border this indicates virtual inheritance.*
- *A solid arrow indicates public inheritance.*
- *A dashed arrow indicates protected inheritance.*
- *A dotted arrow indicates private inheritance.*

The elements in the graphs (...) have the following meaning:

- *A white box indicates a class or struct or file.*
- *A box with a red border indicates a node that has more arrows than are shown! In other words: the graph is truncated with respect to this node. The reason why a graph is sometimes truncated is to prevent images from becoming too large. For the graphs generated with dot doxygen tries to limit the width of the resulting image to 1024 pixels.*
- *A black box indicates that the class' documentation is currently shown.*
- *A dark blue arrow indicates an include relation (for the include dependency graph) or public inheritance (for the other graphs).*

- A dark green arrow indicates protected inheritance.
- A dark red arrow indicates private inheritance.
- A purple dashed arrow indicates a "usage" relation, the edge of the arrow is labeled with the variable(s) responsible for the relation. Class A uses class B, if class A has a member variable m of type C, where B is a subtype of C (e.g. C could be B, B*, T*).

1.7 Support

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

zplane.development

holsteinische str. 39-42

aufgang 8

D-12161 berlin

germany

fon: +49.30.854 09 15.0

fax: +49.30.854 09 15.5

@: info@zplane.de

2 espace realtime convolution SDK Class Index

2.1 espace realtime convolution SDK Class List

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

EnvelopePoint	13
espaceIf	13

3 espace realtime convolution SDK File Index

3.1 espace realtime convolution SDK File List

Here is a list of all documented files with brief descriptions:

espaceAPI.h (Interface of the espaceIf class)	20
--	----

4 espace realtime convolution SDK Class Documentation

4.1 `_EnvelopePoint_` Struct Reference

```
#include <espaceAPI.h>
```

4.1.1 Detailed Description

format for one point of the envelope

Definition at line 58 of file `espaceAPI.h`.

Public Attributes

- float `fAmplitudeIndB`
amplitude value in dBFS (0dB equals no alteration)
- float `fTimeInPercent`
position of envelope point in percent

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

- [espaceAPI.h](#)

4.2 `espaceIf` Class Reference

```
#include <espaceAPI.h>
```

4.2.1 Detailed Description

CLASS

This class provides the interface for espace realtime parametric convolution.

Definition at line 135 of file `espaceAPI.h`.

Public Member Functions

- virtual `zERROR Initialize` (int `iSampleRate`, int `iNumOfChannels`, int `iBlockSize`)=0
- virtual `zERROR Process` (const float `**ppfInputData`, float `**ppfOutputData`, int `iNumOfFrames`)=0
- virtual `zERROR Reset` ()=0

- virtual zERROR [SetImpulseResponse](#) (const float **ppfImpulseResponse, int iNumOfIRFrames, int iNumOfIRChannels, int iIRSampleRate)=0
- virtual zERROR [GetProcessImpulseResponse](#) (float **ppfImpulseResponse, int *piNumOfIRFrames, int *piNumOfIRChannels)=0
- virtual zERROR [SetParameter](#) (int iIndex, float fValue)=0
- virtual float [GetParameter](#) (int iIndex)=0
- virtual zERROR [SetStretchnPitch](#) (float fStretchFactor, float fPitchFactor, [_IRPart_](#) ePart)=0
- virtual zERROR [CreateFilter](#) (void **phFilterHandle, [_FilterTypes_](#) eType, [_IRPart_](#) ePart)=0
- virtual zERROR [DestroyFilter](#) (void **phFilterHandle)=0
- virtual zERROR [SetFilterParameter](#) (void *phFilterHandle, float fFilterFreq, float fFilterGain=0, float fFilterQ=1)=0
- virtual zERROR [GetFilterParameter](#) (void *phFilterHandle, float *pfFilterFreq, float *pfFilterGain=0, float *pfFilterQ=0)=0
- virtual zERROR [SetEnvelope](#) ([_EnvelopePoint_](#) **ppstEnvelopePoints, int iNumOfEnvelopePoints)=0
- virtual int [GetEnvelope](#) ([_EnvelopePoint_](#) **pstEnvelopePoints=0)=0
- virtual void [ApplyParameterChanges](#) ([PFUNCCallback](#) pCallbackWhenReady=0, void *pUserData=0)=0

Static Public Member Functions

- zERROR [CreateInstance](#) (espaceIf *&pCespace)
- zERROR [DestroyInstance](#) (espaceIf *&pCespace)

4.2.2 Member Function Documentation

4.2.2.1 virtual void espaceIf::ApplyParameterChanges ([PFUNCCallback](#) *pCallbackWhenReady* = 0, void **pUserData* = 0) [pure virtual]

this function has to be called after the call of the functions [SetParameter](#)/[SetEnvelope](#)/[SetFilterParameter](#)/[SetStretchnPitch](#) to apply the parameter changes

Parameters:

pCallbackWhenReady : pointer to callback function to be called if the parameters are applied

pUserData : pointer to any user data that can be returned with the callback function

Returns:

virtual void :

4.2.2.2 virtual zERROR espaceIf::CreateFilter (void ** *phFilterHandle*, [_FilterTypes_](#) *eType*, [_IRPart_](#) *ePart*) [pure virtual]

add a new filter either to early reflections or to late reverberation (up to ten filters per IR part are allowed), this function should be followed by a call of ::SetFilterParameter

Parameters:

- **phFilterHandle* : handle to the filter, to be written
- eType* : the type of the filter to be applied see [_FilterTypes_](#)
- ePart* : the part of the IR where the filter should be applied (early reflections or late reverberation)

Returns:

virtual zERROR : 0 when no error

4.2.2.3 zERROR espaceIf::CreateInstance ([espaceIf](#) *& *pCespace*) [static]

creates a new instance of espace

Parameters:

- pCespace* : handle to new instance, to be written

Returns:

static zERROR : 0 when no error

4.2.2.4 virtual zERROR espaceIf::DestroyFilter (void ** *phFilterHandle*)
[pure virtual]

destroy a filter

Parameters:

- **phFilterHandle* : handle to the filter to be destroyed

Returns:

virtual zERROR : 0 when no error

4.2.2.5 zERROR espaceIf::DestroyInstance ([espaceIf](#) *& *pCespace*)
[static]

destroys an instance of espace

Parameters:

- pCespace* : handle to the instance to be destroyed

Returns:

static zERROR : 0 when no error

4.2.2.6 virtual int espaceIf::GetEnvelope ([_EnvelopePoint](#) ***pstEnvelopePoints* = 0) [pure virtual]

returns the current envelope

Parameters:

****pstEnvelopePoints** : array of pointers to structures containing amplitude and time points of envelope, if 0, only the number of envelope points is returned

Returns:

virtual int : number of envelope points

4.2.2.7 virtual zERROR espaceIf::GetFilterParameter (void * *phFilterHandle*, float * *pfFilterFreq*, float * *pfFilterGain* = 0, float * *pfFilterQ* = 0) [pure virtual]

returns the parameters of the filter

Parameters:

***phFilterHandle** : handle to the filter

pfFilterFreq : depending on the selected filter type the cut-off frequency or the mid frequency in Hz

pfFilterGain : gain of the filter in dB, only set if required by the filter type

pfFilterQ : Q of the filter, only set if required by the filter type

Returns:

virtual zERROR : 0 when no error

4.2.2.8 virtual float espaceIf::GetParameter (int *iIndex*) [pure virtual]

return value of parameter with index iIndex

Parameters:

iIndex : parameter index

Returns:

virtual float : value of parameter with index iIndex

4.2.2.9 virtual zERROR espaceIf::GetProcessImpulseResponse (float ** *ppfImpulseResponse*, int * *piNumOfIRFrames*, int * *piNumOfIRChannels*) [pure virtual]

get the current impulse response (e.g. for display)

Parameters:

****ppfImpulseResponse** : double pointer to the impulse response buffer of samples [channels][samples], to be written. NOTE THAT THE MEMORY HAS TO BE ALLOCATED BY THE USER!!!

**piNumOfIRFrames* : number of impulse response frames, to be written
**piNumOfIRChannels* : number of impulse response channels, to be written

Returns:

virtual zERROR : 0 when no error

4.2.2.10 virtual zERROR espaceIf::Initialize (int iSampleRate, int iNumOfChannels, int iBlockSize) [pure virtual]

instance initialization and allocation of IR-independent memory

Parameters:

iSampleRate : sample rate of instance in Hz
iNumOfChannels : number of channels of instance
iBlockSize : processing block size in frames

Returns:

virtual zERROR : 0 when no error

4.2.2.11 virtual zERROR espaceIf::Process (const float ** ppfInputData, float ** ppfOutputData, int iNumOfFrames) [pure virtual]

process function (does the actual convolution), if the instance is not ready for processing, the output buffer is set to 0

Parameters:

***ppfInputData* : double pointer to the input buffer of samples
[channels][samples]
***ppfOutputData* : double pointer to the output buffer of samples
[channels][samples], may be the same as the input buffer
iNumOfFrames : number of input frames, must not exceed 16384

Returns:

virtual zERROR : 0 when no error

4.2.2.12 virtual zERROR espaceIf::Reset () [pure virtual]

reset processing buffers (but not impulse response/settings)

Parameters:

none

Returns:

virtual zERROR : 0 when no error

4.2.2.13 virtual zERROR espaceIf::SetEnvelope (EnvelopePoint ** *ppstEnvelopePoints*, int *iNumOfEnvelopePoints*) [pure virtual]

set an envelope for the (whole) IR

Parameters:

*****ppstEnvelopePoints*** : array of pointers to structures containing amplitude and time points of envelope

iNumOfEnvelopePoints : number of array elements

Returns:

virtual zERROR :

4.2.2.14 virtual zERROR espaceIf::SetFilterParameter (void * *phFilterHandle*, float *fFilterFreq*, float *fFilterGain* = 0, float *fFilterQ* = 1) [pure virtual]

set filter parameter for a special filter, this function has to be followed by a call of the function ::ApplyParameterChanges, except SetParameter/SetStretchnPitch/SetEnvelope/SetFilterParameter is called afterwards with other parameters

Parameters:

****phFilterHandle*** : handle to the filter

fFilterFreq : depending on the selected filter type the cut-off frequency or the mid frequency in Hz

fFilterGain : gain of the filter in dB, only to be set if required by the filter type

fFilterQ : Q of the filter, only to be set if required by the filter type

Returns:

virtual zERROR : 0 when no error

4.2.2.15 virtual zERROR espaceIf::SetImpulseResponse (const float ** *ppfImpulseResponse*, int *iNumOfIRFrames*, int *iNumOfIRChannels*, int *iIRSampleRate*) [pure virtual]

set new impulse response for processing

Parameters:

*****ppfImpulseResponse*** : double pointer to the impulse response buffer of samples [channels][samples]

iNumOfIRFrames : number of frames

iNumOfIRChannels : number of IR channels (has to equal *iNumOfChannels* or 1)

iIRSampleRate : sample rate of IR (IR will be converted if *iIRSampleRate* != *iSampleRate*)

Returns:

virtual zERROR : 0 when no error

4.2.2.16 virtual zERROR espaceIf::SetParameter (int *iIndex*, float *fValue*)
[pure virtual]

set parameter with index *iIndex* to value *fValue*, this function has to be followed by a call of the function `::ApplyParameterChanges`, except `SetParameter/SetStretchnPitch/SetEnvelope/SetFilterParameter` is called afterwards with other parameters

Parameters:

iIndex : parameter index, enum [_Parameter_Indices_](#)
fValue : value of parameter

Returns:

virtual zERROR : 0 when no error

4.2.2.17 virtual zERROR espaceIf::SetStretchnPitch (float *fStretchFactor*, float *fPitchFactor*, [_IRPart_ ePart](#)) [pure virtual]

set stretch and pitch factor for early reflections or late reverberation, this function has to be followed by a call of the function `::ApplyParameterChanges`, except `SetParameter/SetStretchnPitch/SetEnvelope/SetFilterParameter` is called afterwards with other parameters

Parameters:

fStretchFactor : stretch factor in percent (1.0 means no alteration)
fPitchFactor : pitch factor in percent (1.0 means no alteration)
ePart : part of the IR (early reflections or late reverberation)

Returns:

virtual zERROR : 0 when no error

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

- [espaceAPI.h](#)

5 espace realtime convolution SDK File Documentation

5.1 docugen.txt File Reference

5.1.1 Detailed Description

source documentation main file

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

5.2 espaceAPI.h File Reference

5.2.1 Detailed Description

interface of the [espaceIf](#) class.

:

Definition in file [espaceAPI.h](#).

Classes

- struct [_EnvelopePoint_](#)
- class [espaceIf](#)

Typedefs

- typedef void(* [PFUNCallback](#))(void *pUserData)

Enumerations

- enum [_FilterTypes_](#) {
[FilterTypeLowPass](#), [FilterTypeHighPass](#), [FilterTypeBandPass](#), [FilterTypePeak](#),
[FilterTypeLowShelving](#), [FilterTypeHighShelving](#) }
- enum [_IRPart_](#) { [ER](#), [LR](#) }
- enum [_Parameter_Indices_](#) {
[kParamIndexStart](#) = 0, [kParamInformalStart](#) = 0, [kParamProcessLatency](#), [kParamLengthOfOrigIR](#),
[kParamLengthOfCurrentIR](#), [kParamReverberationTime](#), [kParamNumOfTailSamples](#), [kParamStretchER](#),
[kParamPitchER](#), [kParamStretchLR](#), [kParamPitchLR](#), [kNumParamsInformal](#),
[kParamSettableStart](#) = 100, [kParamXPoint](#), [kParamWetness](#), [kParamPreDelay](#),
[kParamAutoVolumeScaleEnabled](#), [kParamReverseIR](#), [kParamPreserveLength](#),
[kParamBypass](#),
[kParamBlockSize](#), [kParamInputMode](#), [kParamXFadeLength](#), [kParamResLPCutOff](#),
[kParamResLPResonance](#), [kParamResLPModFreq](#), [kParamResLPModDepth](#), [kParamIndexEnd](#),
[kNumParamsSettable](#) = ([kParamResLPModDepth](#) + 1 - [kParamSettableStart](#)) }

5.2.2 Typedef Documentation

5.2.2.1 typedef void(* [PFUNCallback](#))(void *pUserData)

this is the callback function prototype that can be called when all parameter changes are applied Definition at line 54 of file [espaceAPI.h](#).

5.2.3 Enumeration Type Documentation

5.2.3.1 enum [_FilterTypes_](#)

These are the allowed filter types applicable to either early reflections or late reverberation

Enumeration values:

- FilterTypeLowPass* standard two pole low pass filter
- FilterTypeHighPass* standard two pole high pass filter
- FilterTypeBandPass* standard two pole band pass filter
- FilterTypePeak* peak filter
- FilterTypeLowShelving* low shelving filter
- FilterTypeHighShelving* high shelving filter

Definition at line 66 of file espaceAPI.h.

5.2.3.2 enum [_IRPart_](#)

The impulse response (IR) is split in these distinct parts

Enumeration values:

- ER* early reflections part (first part of IR)
- LR* late reverberation part (second part of IR)

Definition at line 78 of file espaceAPI.h.

5.2.3.3 enum [_Parameter_Indices_](#)

this enumeration holds the defines for all parameter indices

Enumeration values:

- kParamIndexStart* not to be used
- kParamInformalStart* not to be used
- kParamProcessLatency* not settable (only informative), in s
- kParamLengthOfOrigIR* not settable (only informative), in s
- kParamLengthOfCurrentIR* not settable (only informative), in s
- kParamReverberationTime* not settable (only informative), RT60 in s
- kParamNumOfTailSamples* not settable (only informative), length of tail in samples (for last processing block)
- kParamStretchER* not settable (only informative), in percent, use function SetStretchnPitch to set
- kParamPitchER* not settable (only informative), in percent, use function SetStretchnPitch to set
- kParamStretchLR* not settable (only informative), in percent, use function SetStretchnPitch to set

- kParamPitchLR*** not settable (only informative), in percent, use function SetStretchnPitch to set
- kNumParamsInformal*** not to be used
- kParamSettableStart*** not to be used
- kParamXPoint*** crosspoint between ER and LR in percent (range 0...1, default: IR dependent)
- kParamWetness*** wetness of output signal in percent (range 0...1, default: 1)
- kParamPreDelay*** predelay in ms (range 0...300, default: 0)
- kParamAutoVolumeScaleEnabled*** enabling/disabling of automatic volume scale (boolean, range 0 or 1, default: 1)
- kParamReverseIR*** bool if whole IR should be flipped (boolean, range 0 or 1, default: 0)
- kParamPreserveLength*** bool if unnecessary data should be truncated or not (boolean, range 0 or 1, default: 1)
- kParamBypass*** bool if input signal is bypassed (boolean, range 0 or 1, default: 0)
- kParamBlockSize*** length of process blocks in frames
- kParamInputMode*** bool if input is downmixed to mono or not (boolean, range 0 or 1, default: 0 equals no downmix)
- kParamXFadeLength*** length of crossfade between ER and LR in s
- kParamResLPCutOff*** cut-off frequency of resonator low pass to be applied to the output signal in Hz
- kParamResLPResonance*** resonance of resonator low pass to be applied to the output signal (value between 0 and 4)
- kParamResLPModFreq*** modulation frequency of resonator low pass to be applied to the output signal (< 20Hz)
- kParamResLPModDepth*** second cut-off frequency of resonator low pass to be applied to the output signal, the modulation varies between kParamResLPCutOff and this parameter
- kParamIndexEnd*** not to be used
- kNumParamsSettable*** not to be used

Definition at line 86 of file espaceAPI.h.