libear

Release 0.9

EBU

May 03, 2023

CONTENTS

libear is a C++14 library to render ADM content according to [bs2127]. It is not a complete application, but provides components to calculate gains and apply them to audio, for embedding into applications which need to render ADM content.

Rendering of ADM content is performed by two distinct processes:

- Calculating gains to apply to the input audio samples, as described in *Calculating Gains*.
- Applying those gains to the input audio samples to produce output audio samples, as described in *DSP*.

To get started, check out the *Installation* instructions.

CHAPTER

ONE

SUPPORT

DirectSpeakers, Objects and HOA typeDefinitions are supported, though the following parameters/features are currently not implemented:

Objects:

- Cartesian positions, or `cartesian == true`
- `divergence`
- `zoneExclusion`
- `channelLock`
- `screenRef`
- `screenEdgeLock`

DirectSpeakers:

- Cartesian positions
- `screenEdgeLock`

All types:

- `M-SC` and `M+SC` loudspeakers with azimuths wider than 25 degrees

libear aims to be a complete renderer implementation; these deficiencies will be addressed in future releases.

libear does not include functionality to read BW64 files or parse ADM XML data; for that functionality we recommend using libbw64 and libadm.

CHAPTER

TWO

CREDITS

libear is a joint development between IRT and BBC R&D.

CHAPTER

THREE

LICENSE

```
Copyright 2019 The libear Authors

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

3.1 Installation

3.1.1 Dependencies

- compiler with C++14 support
- Boost header libraries (version 1.57 or later)
  - Boost.Optional
  - Boost.Variant
- CMake build system (version 3.5 or later)

3.1.2 Installation

To manually install the library you have to recursively clone the git repository and then use the CMake build system to build and install it.

```
git clone --recursive https://github.com/ebu/libear.git
cd libear
mkdir build && cd build
cmake ..
make
make install
```

3.1.3 Use from CMake Projects

As the library uses CMake as a build system it is really easy to set up and use if your project does too. Assuming you have installed the library, the following code shows a complete CMake example to compile a program which uses the libear.

```
cmake_minimum_required(VERSION 3.5)
project(libear_example VERSION 1.0.0 LANGUAGES CXX)

find_package(ear REQUIRED)

add_executable(example example.cpp)
target_link_libraries(example PRIVATE ear)
```

3.1.4 Use as a Subproject

If you prefer not to install the library on your system you can also use the library as a CMake subproject. Just add the folder containing the repository to your project (for example, by using a git submodule) and you can use the ear target:

```
cmake_minimum_required(VERSION 3.5)
project(libear_example VERSION 1.0.0 LANGUAGES CXX)

add_subdirectory(submodules/libear)

add_executable(example example.cpp)
target_link_libraries(example PRIVATE ear)
```

Note: If `libear` is used as a CMake subproject the default values of the options

- `EAR_UNIT_TESTS`
- `EAR_EXAMPLES`
- `EAR_PACKAGE_AND_INSTALL`

are automatically set to `FALSE`.

3.2 Calculating Gains

To calculate gains for an ADM item, the ADM metadata is transferred into a `TypeMetadata` struct for that type, which is passed to the `::calculate` method of a `GainCalculator` object instantiated with the desired loudspeaker layout in order to calculate the gains. Both types are dependant on the ADM `typeDefinition`, and are described below.

`TypeMetadata` structures (and sub-structures) for each type are defined in file_ear_metadata.hpp, while `GainCalculator` classes are defined in file_ear_ear.hpp.

The mapping between ADM metadata and `TypeMetadata` structures must be performed by the user of the library. The basic mapping is given in the documentation for each `TypeMetadata` structure (linked below), but for more details see [bs2127] section 5.2.

Errors may be returned by throwing exceptions during `GainCalculator` constructor or `::calculate` calls, while warnings may be returned from `::calculate` through the `warning_cb` parameter; see *Error Handling* for details.

For information on loudspeaker layouts, see *Loudspeaker Layouts*.

3.2.1 DirectSpeakers

DirectSpeakers metadata is represented by the `DirectSpeakersTypeMetadata`, and gains are calculated by `GainCalculatorDirectSpeakers`.

3.2.2 Objects

Objects metadata is represented by the `ObjectsTypeMetadata`, and gains are calculated by `GainCalculatorObjects`.

Two gain vectors are produced by `GainCalculatorObjects::calculate()`, `directGains` and `diffuseGains`. To apply these, see *Rendering Objects*.

3.2.3 HOA

HOA metadata is represented by the `HOATypeMetadata`, and a decode matrix is calculated by `GainCalculatorHOA`.

3.3 DSP

This library does not provide complete DSP paths to render ADM content, but does contain some components which can be used to do so. DSP components are defined in *Namespace ear::dsp*.

3.3.1 FFT interface

`BlockConvolver` objects use a user-provided FFT implementation. These are provided by implementing `FFTImpl<float>` (and therefore `FFTPlan<float>` and `FFTWorkBuf`) for the FFT library you wish to use, and passing an instance to `BlockConvolver::Context::Context()`.

An implementation for KISS FFT is provided by default, and may be obtained by calling `get_fft_kiss()`. The implementation of this (in `src/fft_kiss.cpp`) may be a useful example to show how the FFT interface should be implemented.

3.3.2 Rendering DirectSpeakers

The gains calculated for `DirectSpeakers` channels using the `GainCalculatorDirectSpeakers` should be applied directly to the input audio channel to produce the output audio channels. `DirectSpeakers` metadata should not be dynamic (there should be a single `audioBlockFormat` in each `audioChannelFormat`), so gains should not be interpolated inside blocks, though should be interpolated if metadata is changed by the user.

This may be applied using the `GainInterpolator` with `LinearInterpVector`.

3.3.3 Rendering Objects

The audio processing for Objects content is defined in [bs2127] section 7.1. The structure used is as in Fig. 3.1.

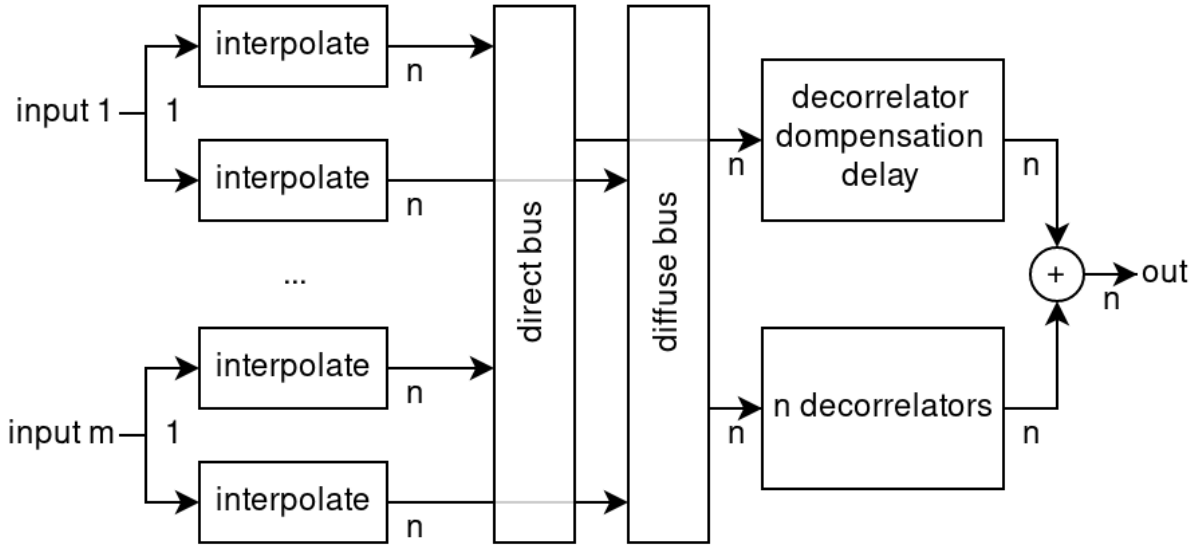

Fig. 3.1: Signal processing for m Objects with n output channels

This can be built from the following components:

- *GainInterpolator* with *LinearInterpVector* to interpolate and apply the gains to the incoming audio (the *interp* blocks in Fig. 3.1).
- *DelayBuffer<float>* with *decorrelatorCompensationDelay()* samples delay to compensate for the decorrelator delays.
- `BlockConvolver` objects with filters calculated using *designDecorrelators()* to decorrelate the signals.
- *VariableBlockSizeAdapter<float>* to allow the use of `BlockConvolver` objects with variable-size sample blocks. This could be used to wrap just the `BlockConvolver::process()` calls, or the whole processing chain (recommended). If only the block convolvers are adapted, then the compensation delay will need to be increased by *VariableBlockSizeAdapter::get_delay()* samples.

3.3.4 Rendering HOA

As with `DirectSpeakers`, `HOA` metadata should not be dynamic, so the calculated matrices can be applied directly to the input audio.

The decode matrices calculated for `HOA` channels using the *GainCalculatorHOA* should be applied directly to the input audio channels to produce the output audio channels. As with `DirectSpeakers`, `HOA` metadata should not be dynamic (there should be a single `audioBlockFormat` in each `audioChannelFormat`), so gains should not be interpolated inside blocks, though should be interpolated if metadata is changed by the user.

This may be applied using the *GainInterpolator* with *LinearInterpMatrix*.

3.4 Loudspeaker Layouts

An output loudspeaker layout is represented by *Layout*, which contains a name, a list of *Channel* objects, and the reference *Screen*.

Loudspeaker layouts specified in [bs2051] are supported, including positions within the ranges specified. The function *getLayout()* is therefore provided to obtain a *Layout* object given the layout name, e.g. `4+5+0`.

When using layouts with non-nominal positions, the *Channel::polarPositionNominal()* must match the position specified in Table 1 in [bs2051], and the *Channel::polarPosition()* must meet the specified constraints, including azimuth/elevation ranges and symmetry requirements.

Non-standard loudspeaker layouts may be used, however their behaviour may change in future versions of the library. Loudspeaker layouts must be similar to those in [bs2051], with 1, 2 or 3 layers and an optional T+000 or UH+180 loudspeaker. Using this functionality requires some understanding of the internals of the renderer, particularly section 6.1.3.1 of [bs2127].

3.5 Metadata Conversion

Functions are provided for converting Objects metadata between polar and Cartesian formats according to [bs2127] section 10.

See the EAR reference documentation for more general information.

To convert positions only, use *pointCartToPolar()* and *pointPolarToCart()*.

To convert positions and extent parameters, use *extentCartToPolar()* and *extentPolarToCart()*.

To convert whole block formats, use *toPolar()* and *toCartesian()*.

3.6 Error Handling

Two classes of error are produced by the library: exceptions and warnings.

3.6.1 Exceptions

Exceptions are thrown by the library for severe errors which prevent the requested operation from being completed. All errors thrown by the library are defined in file_ear_exceptions.hpp

Notably, this library does not yet implement all features of [bs2127]; when such a feature is used, a *not_implemented* will be thrown.

3.6.2 Warnings

Warnings are issued in less severe cases, where the requested operation could still be completed. Warnings are generally issued for user-facing problems, such as errors in metadata, and so should be visible to the user.

Warnings are returned from the library through the `warnings_cb` argument to `::calculate` methods on `GainCalculator` objects (for example *GainCalculatorObjects::calculate()*). Each time a warning is issued, the provided callback will be called with a *Warning* structure, containing the type and message of the warning. By default, *default_warning_cb* is used for this argument, which will print warnings to stderr.

Warning structures are defines in file_ear_warnings.hpp.

3.7 API Reference

3.7.1 Class Hierarchy

3.7.2 File Hierarchy

3.7.3 Full API

Namespaces

Namespace ear

Contents

Namespaces

Classes

- *Struct PolarScreen*
- *Struct PolarSpeakerPosition*
- *Struct ScreenEdgeLock*
- *Struct Warning*
- *Struct ZoneExclusion*
- *Class adm_error*
- *Class Channel*
- *Template Class FFTImpl*
- *Template Class FFTPlan*
- *Class FFTWorkBuf*
- *Class GainCalculatorDirectSpeakers*
- *Class GainCalculatorHOA*
- *Class GainCalculatorObjects*
- *Class internal_error*
- *Class invalid_argument*
- *Class Layout*
- *Class not_implemented*
- *Class unknown_layout*

Functions

- *Function ear::decorrelatorCompensationDelay*
- *Template Function ear::designDecorrelators*
- *Template Function ear::get_fft_kiss*
- *Function ear::getDefaultScreen*
- *Function ear::getLayout*
- *Function ear::loadLayouts*

Typedefs

- *Typedef ear::ExclusionZone*
- *Typedef ear::ObjectDivergence*
- *Typedef ear::Position*
- *Typedef ear::Screen*
- *Typedef ear::SpeakerPosition*
- *Typedef ear::WarningCB*

Variables

- *Variable ear::default_warning_cb*

Namespace ear::conversion

Contents

Classes

- *Struct ExtentParams*

Functions

- *Function ear::conversion::extentCartToPolar*
- *Function ear::conversion::extentPolarToCart*
- *Function ear::conversion::pointCartToPolar*
- *Function ear::conversion::pointPolarToCart*
- *Function ear::conversion::toCartesian*
- *Function ear::conversion::toPolar*

Namespace ear::dsp

Contents

Namespaces

- *Namespace ear::dsp::block_convolver*

Classes

- *Template Struct InterpType*
- *Struct LinearInterpMatrix*
- *Struct LinearInterpSingle*
- *Struct LinearInterpVector*
- *Class DelayBuffer*
- *Template Class GainInterpolator*
- *Template Class PtrAdapterT*
- *Class VariableBlockSizeAdapter*

Typedefs

- *Typedef ear::dsp::PtrAdapter*
- *Typedef ear::dsp::PtrAdapterConst*
- *Typedef ear::dsp::SampleIndex*

Namespace ear::dsp::block_convolver

Contents

Classes

- *Class BlockConvolver*
- *Class Context*
- *Class Filter*

Typedefs

- *Typedef ear::dsp::block_convolver::complex_t*
- *Typedef ear::dsp::block_convolver::real_t*

Classes and Structs

Struct CartesianExclusionZone

- Defined in file_ear_metadata.hpp

Struct Documentation

struct `ear::CartesianExclusionZone`

Public Members

float **minX**

float **maxX**

float **minY**

float **maxY**

float **minZ**

float **maxZ**

std::string **label**

Struct CartesianObjectDivergence

- Defined in file_ear_metadata.hpp

Struct Documentation

struct `ear::CartesianObjectDivergence`

Public Functions

inline **CartesianObjectDivergence**(double divergence = 0.0, double positionRange = 0.0)

Public Members

double **divergence**

double **positionRange**

Struct CartesianPosition

- Defined in file_ear_common_types.hpp

Struct Documentation

struct ear::**CartesianPosition**

Public Functions

inline **CartesianPosition**(double X = 0.0, double Y = 0.0, double Z = 0.0)

Public Members

double **X**

double **Y**

double **Z**

Struct CartesianScreen

- Defined in file_ear_screen.hpp

Struct Documentation

struct ear::**CartesianScreen**

Public Members

double **aspectRatio**

CartesianPosition **centrePosition**

double **widthX**

Struct CartesianSpeakerPosition

- Defined in file_ear_metadata.hpp

Struct Documentation

struct `ear::CartesianSpeakerPosition`

Public Functions

inline **CartesianSpeakerPosition**(double X = 0.0, double Y = 1.0, double Z = 0.0)

Public Members

double **X**

boost::optional<double> **XMin**

boost::optional<double> **XMax**

double **Y**

boost::optional<double> **YMin**

boost::optional<double> **YMax**

double **Z**

boost::optional<double> **ZMin**

boost::optional<double> **ZMax**

ScreenEdgeLock **screenEdgeLock**

Struct ChannelFrequency

- Defined in file_ear_metadata.hpp

Struct Documentation

struct ear::**ChannelFrequency**

Public Members

boost::optional<double> **lowPass** = boost::none

boost::optional<double> **highPass** = boost::none

Struct ChannelLock

- Defined in file_ear_metadata.hpp

Struct Documentation

struct ear::**ChannelLock**

Public Functions

inline **ChannelLock**(bool flag = false, boost::optional<double> maxDistance = boost::none)

Public Members

bool **flag**

boost::optional<double> **maxDistance**

Struct ExtentParams

- Defined in file_ear_conversion.hpp

Struct Documentation

struct ear::conversion::**ExtentParams**
structure for holding extent parameters without using an entire *ObjectsTypeMetadata* object

Public Members

double **width**

double **height**

double **depth**

Struct DirectSpeakersTypeMetadata

- Defined in file_ear_metadata.hpp

Struct Documentation

struct `ear::DirectSpeakersTypeMetadata`

Public Members

std::vector<std::string> **speakerLabels** = {}
contents of the `speakerLabel` tags in this `audioBlockFormat`, in the order given in the AXML

SpeakerPosition **position** = *PolarSpeakerPosition*()
contents of the `position` elements

ChannelFrequency **channelFrequency** = {}
`frequency` information contained in the `audioChannelFormat`

boost::optional<std::string> **audioPackFormatID** = boost::none
`audioPackFormatID` of the `audioPackFormat` directly referencing this channel, for example `AP_00010002`

Template Struct InterpType

- Defined in file_ear_dsp_gain_interpolator.hpp

Struct Documentation

template<typename **PointT**>

struct `ear::dsp::InterpType`
Base type for interpolation types.

Public Types

using **Point** = *PointT*
Type of points on the interpolation curve, for example float, vector, matrix.

Public Static Functions

static inline bool **constant_interp**(const *Point* &a, const *Point* &b)
Are the two points the same (and therefore constant/no interpolation should be used between them)?

static void **apply_interp**(const float *const *in, float *const *out, *SampleIndex* range_start, *SampleIndex* range_end, *SampleIndex* block_start, *SampleIndex* start, *SampleIndex* end, const *Point* &start_point, const *Point* &end_point)
Apply interpolated gains to `in`, writing to `out`.

For example, if an interpolation curve goes from x to y between sample 5 and 15, these calls would occur for the first and second 10-sample blocks:

```
apply_interp(in_a, out_a, 5, 10, 0, 5, 15, x, y);
apply_interp(in_b, out_b, 0, 5, 10, 5, 15, x, y);
```

Parameters

- **in** – input samples
- **out** – output samples
- **range_start** – offset in `in` and `out` to start processing
- **range_end** – offset in `in` and `out` to end processing
- **block_start** – start sample index of this block, i.e. in[0][0]
- **start** – start sample index of interpolation curve
- **end** – end sample index of interpolation curve
- **start_point** – gain values at `start`
- **end_point** – gain values at `end`

static void **apply_constant**(const float *const *in, float *const *out, *SampleIndex* range_start, *SampleIndex* range_end, const *Point* &point)
Apply constnt gain gains to `in`, writing to `out`.

Parameters

- **in** – input samples
- **out** – output samples
- **range_start** – offset in `in` and `out` to start processing
- **range_end** – offset in `in` and `out` to end processing
- **point** – gain values to apply

Struct LinearInterpMatrix

- Defined in file_ear_dsp_gain_interpolator.hpp

Inheritance Relationships

Base Type

- `public ear::dsp::InterpType< std::vector< std::vector< float > > >` (*Template Struct InterpType*)

Struct Documentation

struct `ear::dsp::LinearInterpMatrix` : public ear::dsp::*InterpType*<std::vector<std::vector<float>>>
Linear interpolation with multiple input and output channels, with a matrix of coefficients for use in *GainInterpolator*.

Points contain one vector of per-output gains per input channel.

Public Static Functions

static inline void **apply_interp**(const float *const *in, float *const *out, *SampleIndex* range_start, *SampleIndex* range_end, *SampleIndex* block_start, *SampleIndex* start, *SampleIndex* end, const Point &start_point, const Point &end_point)

static inline void **apply_constant**(const float *const *in, float *const *out, *SampleIndex* range_start, *SampleIndex* range_end, const Point &point)

Struct LinearInterpSingle

- Defined in file_ear_dsp_gain_interpolator.hpp

Inheritance Relationships

Base Type

- `public ear::dsp::InterpType< float >` (*Template Struct InterpType*)

Struct Documentation

struct `ear::dsp::LinearInterpSingle` : public ear::dsp::*InterpType*<float>
Linear interpolation of a single channel for use in *GainInterpolator*.

Public Static Functions

static inline void **apply_interp**(const float *const *in, float *const *out, *SampleIndex* range_start, *SampleIndex* range_end, *SampleIndex* block_start, *SampleIndex* start, *SampleIndex* end, const Point &start_point, const Point &end_point)

static inline void **apply_constant**(const float *const *in, float *const *out, *SampleIndex* range_start, *SampleIndex* range_end, const Point &point)

Struct LinearInterpVector

- Defined in file_ear_dsp_gain_interpolator.hpp

Inheritance Relationships

Base Type

- `public ear::dsp::InterpType< std::vector< float > >` (*Template Struct InterpType*)

Struct Documentation

struct `ear::dsp::LinearInterpVector` : public ear::dsp::*InterpType*<std::vector<float>>
Linear interpolation with one channel in and multiple out for use in *GainInterpolator*.

Public Static Functions

static inline void **apply_interp**(const float *const *in, float *const *out, *SampleIndex* range_start, *SampleIndex* range_end, *SampleIndex* block_start, *SampleIndex* start, *SampleIndex* end, const Point &start_point, const Point &end_point)

static inline void **apply_constant**(const float *const *in, float *const *out, *SampleIndex* range_start, *SampleIndex* range_end, const Point &point)

Struct HOATypeMetadata

- Defined in file_ear_metadata.hpp

Struct Documentation

struct `ear::HOATypeMetadata`
Representation of all audioChannelFormats in a HOA audioPackFormat.

orders and *degrees* must be the same length and must be in the same order as the channels being rendered, such that the `i`th input channel has order `orders[i]` and degree `degrees[i]`.

`normalization`, `nfcRefDist` and `screenRef` may be defined in an `audioBlockFormat` and/or `audioPackFormat`; see the rules in [bs2127] section 5.2.7.3 for details.

Public Members

std::vector<int> **`orders`**
value of the `order` element in the `audioBlockFormat` element of each channel

std::vector<int> **`degrees`**
value of the `degree` element in the `audioBlockFormat` element of each channel

std::string **`normalization`** = std::string("SN3D")

double **`nfcRefDist`** = 0.0

bool **`screenRef`** = false

Screen **`referenceScreen`** = *getDefaultScreen*()
screen specification from the `audioProgrammeReferenceScreen` element of the `audioProgramme` being rendered

Struct ObjectsTypeMetadata

- Defined in file_ear_metadata.hpp

Struct Documentation

struct `ear::ObjectsTypeMetadata`

Public Members

Position **position** = {}

double **width** = 0.0

double **height** = 0.0

double **depth** = 0.0

bool **cartesian** = false
value of the `cartesian` flag; should be the same type as used in position, objectDivergence and zoneExclusion, otherwise expect warnings.

double **gain** = 1.0

double **diffuse** = 0.0

ChannelLock **channelLock** = {}

ObjectDivergence **objectDivergence** = {}

ZoneExclusion **zoneExclusion** = {}

bool **screenRef** = false

Screen **referenceScreen** = *getDefaultScreen*()
screen specification from the `audioProgrammeReferenceScreen` element of the `audioProgramme` being rendered

Struct PolarExclusionZone

- Defined in file_ear_metadata.hpp

Struct Documentation

struct `ear::`**PolarExclusionZone**

Public Members

float **minAzimuth**

float **maxAzimuth**

float **minElevation**

float **maxElevation**

float **minDistance**

float **maxDistance**

std::string **label**

Struct PolarObjectDivergence

- Defined in file_ear_metadata.hpp

Struct Documentation

struct ear::**PolarObjectDivergence**

Public Functions

inline **PolarObjectDivergence**(double divergence = 0.0, double azimuthRange = 45.0)

Public Members

double **divergence**

double **azimuthRange**

Struct PolarPosition

- Defined in file_ear_common_types.hpp

Struct Documentation

struct ear::**PolarPosition**

Public Functions

inline **PolarPosition**(double azimuth = 0.0, double elevation = 0.0, double distance = 1.0)

Public Members

double **azimuth**

double **elevation**

double **distance**

Struct PolarScreen

- Defined in file_ear_screen.hpp

Struct Documentation

struct ear::**PolarScreen**

Public Members

double **aspectRatio**

PolarPosition **centrePosition**

double **widthAzimuth**

Struct PolarSpeakerPosition

- Defined in file_ear_metadata.hpp

Struct Documentation

struct ear::**PolarSpeakerPosition**

Public Functions

inline **PolarSpeakerPosition**(double az = 0.0, double el = 0.0, double dist = 1.0)

Public Members

double **azimuth**

boost::optional<double> **azimuthMin**

boost::optional<double> **azimuthMax**

double **elevation**

boost::optional<double> **elevationMin**

boost::optional<double> **elevationMax**

double **distance**

boost::optional<double> **distanceMin**

boost::optional<double> **distanceMax**

ScreenEdgeLock **screenEdgeLock**

Struct ScreenEdgeLock

- Defined in file_ear_metadata.hpp

Struct Documentation

struct `ear::ScreenEdgeLock`

Public Members

boost::optional<std::string> **horizontal**
`screenEdgeLock` attribute on `position` element with `coordinate="azimuth"` or `coordinate="X"`

boost::optional<std::string> **vertical**
`screenEdgeLock` attribute on `position` element with `coordinate="elevation"` or `coordinate="Z"`

Struct Warning

- Defined in file_ear_warnings.hpp

Struct Documentation

struct `ear::Warning`
A warning message, containing a code and a corresponding message.

The code does not need to be shown when displaying warnings; the message should contain all the information required, the code is just to allow implementations to take action on warnings without matching against the messages.

Public Types

enum **Code**
Values:

enumerator **FREQ_SPEAKERLABEL_LFE_MISMATCH**
LFE indication from frequency element does not match speakerLabel.

enumerator **FREQ_NOT_LFE**
frequency indication present but does not indicate an LFE channel

enumerator **FREQ_IGNORED**
frequency information is not implemented; ignoring

enumerator **HOA_SCREENREF_NOT_IMPLEMENTED**
screenRef for HOA is not implemented; ignoring

enumerator **HOA_NFCREFDIST_NOT_IMPLEMENTED**
nfcRefDist is not implemented; ignoring

Public Members

Code **code**

std::string **message**

Struct ZoneExclusion

- Defined in file_ear_metadata.hpp

Struct Documentation

struct `ear::ZoneExclusion`

Public Members

std::vector<*ExclusionZone*> **zones**

Class adm_error

- Defined in file_ear_exceptions.hpp

Inheritance Relationships

Base Type

- `public invalid_argument`

Class Documentation

class `ear::adm_error` : public *invalid_argument*
thrown if invalid ADM metadata is encountered

Public Functions

inline explicit **adm_error**(const std::string &what)

Class Channel

- Defined in file_ear_layout.hpp

Class Documentation

class ear::**Channel**

Representation of a channel, with a name, real and nominal positions, allowed azimuth and elevation ranges, and an lfe flag.

Public Functions

Channel() = default

Channel(const std::string &name, *PolarPosition* polarPosition, boost::optional<*PolarPosition*> polarPositionNominal = boost::none, boost::optional<std::pair<double, double>> azimuthRange = boost::none, boost::optional<std::pair<double, double>> elevationRange = boost::none, bool isLfe = false)

Parameters

- **name** – *Channel* name.
- **polarPosition** – real speaker location
- **polarPositionNominal** – nominal speaker location, defaults to polar_position
- **azimuthRange** – azimuth range in degrees; allowed range is interpreted as starting at azimuthRange[0], moving anticlockwise to azimuthRange[1]; defaults to the azimuth of polar_nominal_position.
- **elevationRange** – elevation range in degrees; allowed range is interpreted as starting at elevationRange.first, moving up to elevationRange.second defaults to the elevation of polar_nominal_position.
- **isLfe** – flag to indicate an LFE channel

std::string **name**() const

PolarPosition **polarPosition**() const

PolarPosition **polarPositionNominal**() const

std::pair<double, double> **azimuthRange**() const

std::pair<double, double> **elevationRange**() const

bool **isLfe**() const

void **name**(const std::string &name)

void **polarPosition**(*PolarPosition* polarPosition)

void **polarPositionNominal**(const boost::optional<*PolarPosition*> &polarPositionNominal)

void **azimuthRange**(const boost::optional<std::pair<double, double>> &azimuthRange)

void **elevationRange**(const boost::optional<std::pair<double, double>> &elevationRange)

void **isLfe**(bool isLfe)

void **checkPosition**(std::function<void(const std::string&)> callback) const

Class BlockConvolver

- Defined in file_ear_dsp_block_convolver.hpp

Class Documentation

class ear::dsp::block_convolver::**BlockConvolver**

BlockConvolver implements partitioned overlap-add convolution with a fixed block size, with efficient fading between filters.

Public Functions

BlockConvolver(const *Context* &ctx, size_t num_blocks)

Create a *BlockConvolver* given the block size and number of blocks.

Parameters

- **ctx** – *Context* required for transformations.
- **num_blocks** – Maximum number of blocks of any filter used.

BlockConvolver(const *Context* &ctx, const *Filter* &filter, size_t num_blocks = 0)

Create a *BlockConvolver* given the block size and number of blocks.

If filter == nullptr, num_blocks must be specified.

Parameters

- **ctx** – *Context* required for transformations.
- **filter** – Initial filter to be used, or nullptr for no filter.

- **num_blocks** – Maximum number of blocks of any filter used; using 0 will take the number of blocks from the passed filter.

void **process**(const float *in, float *out)
Pass a block of audio through the filter.

Parameters

- **in** – Input samples of length block_size
- **out** – Output samples of length block_size

void **crossfade_filter**(const *Filter* &filter)
Crossfade to a new filter during the next block.

This is equivalent to:

- Creating a new convolver.
- Passing the next block of samples through the old and new convolvers, with the input to the old faded down across the block, and the input to the new faded up across the block. All subsequent blocks are passed through the new filter.
- Mixing the output of the old and new filters for the next num_blocks blocks.

void **fade_down**()
Crossfade to a zero-valued filter.

void **set_filter**(const *Filter* &filter)
Switch to a different filter at the start of the next block.

void **unset_filter**()
Switch to a zero-valued filter at the start of the next block.

Class Context

- Defined in file_ear_dsp_block_convolver.hpp

Class Documentation

class ear::dsp::block_convolver::**Context**
Static data required to perform convolution of a particular block size; may be shared between any number of *BlockConvolver* and *Filter* instances.

Public Functions

Context(size_t block_size, *FFTImpl*<*real_t*> &fft_impl)
Create a *Context* with a given block size.

Parameters

- **block_size** – Block size in samples.
- **fft_impl** – FFT implementation to use.

Class Filter

- Defined in file_ear_dsp_block_convolver.hpp

Class Documentation

class `ear::dsp::block_convolver::Filter`
A filter response which may be shared between many *BlockConvolver* instances.

This stores the pre-transformed filter blocks.

Public Functions

`Filter`(const *Context* &ctx, size_t n, const *real_t* *filter)

size_t `num_blocks`() const
The number of blocks in the filter.

Class DelayBuffer

- Defined in file_ear_dsp_delay_buffer.hpp

Class Documentation

class `ear::dsp::DelayBuffer`
A multi-channel delay buffer.

Public Functions

`DelayBuffer`(size_t nchannels, size_t nsamples)

Parameters

- **`nchannels`** – number of input and output channels
- **`nsamples`** – length of the delay

void `process`(size_t nsamples, const float *const *input, float *const *output)
Process an arbitrary number of samples.

`input` and `output` have `nchannels` channels and `nsamples` samples.

int `get_delay`() const
Get the delay in samples.

Template Class GainInterpolator

- Defined in file_ear_dsp_gain_interpolator.hpp

Class Documentation

template<typename **InterpType**>

class `ear::dsp::`**GainInterpolator**

Gain interpolator, templated over an interpolation type which defines the type of interpolation (linear, cosine etc.), the type of the values to interpolate between (floats, vectors, matrices), and therefore restrictions on the input and output channel sizes.

An interpolation curve is defined by the points in *interp_points*. Each of these is a pair of the sample index and the gain values at that time. These must be sorted in time order. Duplicate times can be used to specify steps.

See *LinearInterpSingle*, *LinearInterpVector* and *LinearInterpMatrix* for possible interpolation types. See *InterpType* for the interface that interpolation types define.

Public Functions

inline void **process**(*SampleIndex* block_start, size_t nsamples, const float *const *in, float *const *out)

Process n samples.

Parameters

- **block_start** – the index of the first sample relative to the sample indices in *interp_points*
- **nsamples** – number of samples in `in` and `out`
- **in** – input samples with a number of channels compatible with the interpolation type and points used
- **out** – output samples with a number of channels compatible with the interpolation type and points used

Public Members

std::vector<std::pair<*SampleIndex*, typename *InterpType*::Point>> **interp_points**

Template Class PtrAdapterT

- Defined in file_ear_dsp_ptr_adapter.hpp

Class Documentation

template<typename **PtrT** = float*>

class `ear::dsp::`**PtrAdapterT**
Adapter from Eigen matrix expressions (and possibly other things) to float**.

Public Functions

inline **PtrAdapterT**(size_t nchannels)

template<typename **T**>
inline void **set_eigen**(*T* &&mat, size_t offset = 0)
Point each pointer at a column of Eigen Matrix expression `matrix`, with an offset of `offset`.

inline *PtrT* ***ptrs**()
Get a pointer to each channel.

PtrAdapterT(const *PtrAdapterT*&) = delete

PtrAdapterT &**operator=**(const *PtrAdapterT*&) = delete

Class VariableBlockSizeAdapter

- Defined in file_ear_dsp_variable_block_size.hpp

Class Documentation

class `ear::dsp::`**VariableBlockSizeAdapter**
Adapt something that processes fixed-size blocks of samples into one that processes variable sized blocks by adding some delay.

This can be used with e.g. BlockConvolver to process arbitrary block lengths. This isn't built into the BlockConvolver, because it's not always necessary, and because this introduces some delay; if we adapt multiple components with this then we can save some delay compared to having it built into each component.

Public Types

using **ProcessFunc** = void(const float *const *in, float *const *out)

Public Functions

VariableBlockSizeAdapter(size_t block_size, size_t num_channels_in, size_t num_channels_out, std::function<*ProcessFunc*> process_func)

Parameters

- **block_size** – number of samples accepted by `process_func`
- **num_channels_in** – number of input channels
- **num_channels_out** – number of output channels
- **process_func** – function to call to process block_size samples

void **process**(size_t nsamples, const float *const *in, float *const *out)
Process nsamples samples.

int **get_delay**() const
The delay introduced by the variable block size processing, not accounting for any delay introduced by the inner process.

Template Class FFTImpl

- Defined in file_ear_fft.hpp

Class Documentation

template<typename **Real**>

class `ear::`**FFTImpl**
An FFT implementation.

Public Functions

virtual std::shared_ptr<*FFTPlan*<*Real*>> **plan**(size_t n_fft) const = 0
Plan to execute an r2c/c2r FFT using this implementation.

Parameters **n_fft** – number of points in the real parts; i.e. the input to transform_forward and the output of transform_reverse. Must be even.

Template Class FFTPlan

- Defined in file_ear_fft.hpp

Class Documentation

template<typename **Real**>

class `ear::`**FFTPlan**
Plan for performing an FFT of a particular size/layout/type; allocated by calling *FFTImpl::plan*.

This is not mutated when transform_* are called, so one plan may be shared between threads.

Public Types

using **Complex** = std::complex<*Real*>

Public Functions

virtual void **transform_forward**(*Real* *input, *Complex* *output, *FFTWorkBuf* &workbuf) const = 0
Execute an r2c forwards transform.

Parameters

- **input** – n_fft input samples
- **output** – n_fft/2+1 output samples containing the first half of the complex frequency components without any packing.
- **workbuf** – temporary buffers allocated with alloc_workbuf

virtual void **transform_reverse**(*Complex* *input, *Real* *output, *FFTWorkBuf* &workbuf) const = 0
Execute an c2r inverse transform.

Parameters

- **input** – n_fft/2+1 input samples, in the same format as given by transform_forward
- **output** – n_fft output samples
- **workbuf** – temporary buffers allocated with alloc_workbuf

virtual std::unique_ptr<*FFTWorkBuf*> **alloc_workbuf**() const = 0
allocate temporary buffers to be used with transform_*

Class FFTWorkBuf

- Defined in file_ear_fft.hpp

Class Documentation

class **FFTWorkBuf**
temporary buffers needed to perform an FFT; allocated by calling *FFTPlan::alloc_workbuf*.

As this contains data which is mutated by the transform functions, this must not be shared between threads.

Class GainCalculatorDirectSpeakers

- Defined in file_ear_gain_calculators.hpp

Class Documentation

class ear::**GainCalculatorDirectSpeakers**
Gain calculator for typeDefinition == "DirectSpeakers".

Public Functions

GainCalculatorDirectSpeakers(const *Layout* &layout, std::map<std::string, std::string> additionalSubstitutions = {})

template<typename **T**>
void **calculate**(const *DirectSpeakersTypeMetadata* &metadata, std::vector<*T*> &gains, const *WarningCB* &warning_cb = *default_warning_cb*)
Calculate gains for metadata.

`gains` contains per-loudspeaker gains to render this channel.

Class GainCalculatorHOA

- Defined in file_ear_gain_calculators.hpp

Class Documentation

class ear::**GainCalculatorHOA**
Gain calculator for typeDefinition == "HOA".

Public Functions

GainCalculatorHOA(const *Layout* &layout)

template<typename **T**>
void **calculate**(const *HOATypeMetadata* &metadata, std::vector<std::vector<*T*>> &gains, const *WarningCB* &warning_cb = *default_warning_cb*)
Calculate a decode matrix for metadata.

Gains contains one vector of per-loudspeaker gains per input channel, and must be the right size before calling.

Class GainCalculatorObjects

- Defined in file_ear_gain_calculators.hpp

Class Documentation

class ear::**GainCalculatorObjects**
Gain calculator for typeDefinition == "Objects".

Public Functions

GainCalculatorObjects(const *Layout* &layout)

template<typename **T**>
void **calculate**(const *ObjectsTypeMetadata* &metadata, std::vector<*T*> &directGains, std::vector<*T*> &diffuseGains, const *WarningCB* &warning_cb = *default_warning_cb*)
Calculate gains for metadata.

`directGains` and `diffuseGains` contains per-loudspeaker gains to render this channel.

To apply these gains:

- `directGains` are applied to this channel, and summed with other objects into a n-channel direct bus
- `diffuseGains` are applied to this channel, and summed with other objects into a n-channel diffuse bus
- each channel in the diffuse bus is processed with the corresponding FIR filter given by *designDecorrelators()*
- each channel in the direct bus is delayed by *decorrelatorCompensationDelay()* samples to compensate for the delay through the decorrelation filters
- the output of the decorrelation filters and delays are mixed together to form the output

Class internal_error

- Defined in file_ear_exceptions.hpp

Inheritance Relationships

Base Type

- `public runtime_error`

Class Documentation

class ear::**internal_error** : public runtime_error
thrown for errors inside the library, which should not have occurred given any inputs.

This can be caused by an error in the library itself (please report it!) or something going wrong while building the library. This is thrown by ear_assert.

Public Functions

inline explicit **internal_error**(const std::string &what)

Class invalid_argument

- Defined in file_ear_exceptions.hpp

Inheritance Relationships

Base Type

- `public invalid_argument`

Class Documentation

class ear::**invalid_argument** : public *invalid_argument*
thrown if other invariants on parameters are not met

Public Functions

inline explicit **invalid_argument**(const std::string &what)

Class Layout

- Defined in file_ear_layout.hpp

Class Documentation

class ear::Layout
Representation of a loudspeaker layout, with a name and a list of channels.

Public Functions

Layout(std::string name = "", std::vector<*Channel*> channels = std::vector<*Channel*>(), boost::optional<*Screen*> screen = *getDefaultScreen*())

std::string **name**() const

std::vector<*Channel*> &**channels**()

std::vector<*Channel*> **channels**() const

boost::optional<*Screen*> **screen**() const

void **name**(std::string name)

void **screen**(boost::optional<*Screen*> screen)

Layout **withoutLfe**() const

std::vector<bool> **isLfe**() const

std::vector<std::string> **channelNames**() const

void **checkPositions**(std::function<void(const std::string&)> callback) const

Channel **channelWithName**(const std::string &name) const

boost::optional<int> **indexForName**(const std::string &name) const

std::vector<*PolarPosition*> **positions**() const

std::vector<*PolarPosition*> **nominalPositions**() const

Class not_implemented

- Defined in file_ear_exceptions.hpp

Inheritance Relationships

Base Type

- `public runtime_error`

Class Documentation

class `ear::not_implemented` : public runtime_error
thrown if features are used which are not yet implemented

Public Functions

inline explicit **not_implemented**(const std::string &what)

Class unknown_layout

- Defined in file_ear_exceptions.hpp

Inheritance Relationships

Base Type

- `public invalid_argument`

Class Documentation

class `ear::unknown_layout` : public *invalid_argument*
thrown if an unknown loudspeaker layout is requested

Public Functions

inline explicit **unknown_layout**(const std::string &what)

Functions

Function ear::conversion::extentCartToPolar

- Defined in file_ear_conversion.hpp

Function Documentation

std::pair<*PolarPosition*, *ExtentParams*> `ear::conversion::`**`extentCartToPolar`**(const *CartesianPosition* &pos, const *ExtentParams* &extent)

convert a Cartesian position and extent parameters to polar

This corresponds to `ear.core.objectbased.conversion.extent_cart_to_polar()`.

Function ear::conversion::extentPolarToCart

- Defined in file_ear_conversion.hpp

Function Documentation

std::pair<*CartesianPosition*, *ExtentParams*> `ear::conversion::`**`extentPolarToCart`**(const *PolarPosition* &pos, const *ExtentParams* &extent)

convert a polar position and extent parameters to Cartesian

This corresponds to `ear.core.objectbased.conversion.extent_polar_to_cart()`.

Function ear::conversion::pointCartToPolar

- Defined in file_ear_conversion.hpp

Function Documentation

PolarPosition `ear::conversion::`**`pointCartToPolar`**(const *CartesianPosition* &pos)

convert a Cartesian position to polar

This corresponds to `ear.core.objectbased.conversion.point_cart_to_polar()`.

Function ear::conversion::pointPolarToCart

- Defined in file_ear_conversion.hpp

Function Documentation

CartesianPosition ear::conversion::**pointPolarToCart**(const *PolarPosition* &pos)
convert a polar position to Cartesian

This corresponds to ear.core.objectbased.conversion.point_polar_to_cart().

Function ear::conversion::toCartesian

- Defined in file_ear_conversion.hpp

Function Documentation

void ear::conversion::**toCartesian**(*ObjectsTypeMetadata* &otm)
in-place conversion of Objects metadata to Cartesian

The cartesian flag is ignored, and the type of the position is used to determine whether the metadata is Cartesian or polar. If the metadata is polar, then the position and extent parameters are converted to Cartesian, and the cartesian flag is set.

This corresponds to ear.core.objectbased.conversion.to_cartesian().

Function ear::conversion::toPolar

- Defined in file_ear_conversion.hpp

Function Documentation

void ear::conversion::**toPolar**(*ObjectsTypeMetadata* &otm)
in-place conversion of Objects metadata to polar

The cartesian flag is ignored, and the type of the position is used to determine whether the metadata is Cartesian or polar. If the metadata is Cartesian, then the position and extent parameters are converted to polar, and the cartesian flag is cleared.

This corresponds to ear.core.objectbased.conversion.to_polar().

Function ear::decorrelatorCompensationDelay

- Defined in file_ear_decorrelate.hpp

Function Documentation

int ear::**decorrelatorCompensationDelay**()
Get the delay length needed to compensate for decorrelators.

Returns Delay length in samples.

Template Function ear::designDecorrelators

- Defined in file_ear_decorrelate.hpp

Function Documentation

template<typename **T** = float>
std::vector<std::vector<*T*>> ear::**designDecorrelators**(*Layout* layout)
Design one filter for each channel in layout.

Parameters **layout** – *Layout* to design for; channel names are used to allocate filters to channels.

Returns Decorrelation filters.

Template Function ear::get_fft_kiss

- Defined in file_ear_fft.hpp

Function Documentation

template<typename **Real**>
FFTImpl<*Real*> &ear::**get_fft_kiss**()
Get a KISS FFT implementation for a particular type.

This is always available.

Function ear::getDefaultScreen

- Defined in file_ear_screen.hpp

Function Documentation

Screen ear::**getDefaultScreen**()

Function ear::getLayout

- Defined in file_ear_bs2051.hpp

Function Documentation

Layout ear::**getLayout**(const std::string &name)
Get a layout given its ITU-R BS.2051 name (e.g. 4+5+0).

Function ear::loadLayouts

- Defined in file_ear_bs2051.hpp

Function Documentation

std::vector<*Layout*> ear::**loadLayouts**()
Get all ITU-R BS.2051 layouts.

Variables

Variable ear::default_warning_cb

- Defined in file_ear_warnings.hpp

Variable Documentation

const *WarningCB* ear::**default_warning_cb**
default warning callback which prints to stderr with the prefix libear: warning:

Typedefs

Typedef ear::dsp::block_convolver::complex_t

- Defined in file_ear_dsp_block_convolver.hpp

Typedef Documentation

using ear::dsp::block_convolver::**complex_t** = std::complex<*real_t*>
Type for complex data.

Typedef ear::dsp::block_convolver::real_t

- Defined in file_ear_dsp_block_convolver.hpp

Typedef Documentation

using ear::dsp::block_convolver::**real_t** = float
Type for real data (float).

Typedef ear::dsp::PtrAdapter

- Defined in file_ear_dsp_ptr_adapter.hpp

Typedef Documentation

using ear::dsp::**PtrAdapter** = *PtrAdapterT*<float*>

Typedef ear::dsp::PtrAdapterConst

- Defined in file_ear_dsp_ptr_adapter.hpp

Typedef Documentation

using ear::dsp::**PtrAdapterConst** = *PtrAdapterT*<const float*>

Typedef ear::dsp::SampleIndex

- Defined in file_ear_dsp_gain_interpolator.hpp

Typedef Documentation

using ear::dsp::**SampleIndex** = long int
Type used to index into sample buffers.

Typedef ear::ExclusionZone

- Defined in file_ear_metadata.hpp

Typedef Documentation

using ear::**ExclusionZone** = boost::variant<*PolarExclusionZone*, *CartesianExclusionZone*>

Typedef ear::ObjectDivergence

- Defined in file_ear_metadata.hpp

Typedef Documentation

using ear::**ObjectDivergence** = boost::variant<*PolarObjectDivergence*, *CartesianObjectDivergence*>

Typedef ear::Position

- Defined in file_ear_common_types.hpp

Typedef Documentation

using ear::**Position** = boost::variant<*CartesianPosition*, *PolarPosition*>

Typedef ear::Screen

- Defined in file_ear_screen.hpp

Typedef Documentation

using ear::**Screen** = boost::variant<*PolarScreen*, *CartesianScreen*>

Typedef ear::SpeakerPosition

- Defined in file_ear_metadata.hpp

Typedef Documentation

using `ear::SpeakerPosition` = boost::variant<*PolarSpeakerPosition*, *CartesianSpeakerPosition*>

Typedef ear::WarningCB

- Defined in file_ear_warnings.hpp

Typedef Documentation

using `ear::WarningCB` = std::function<void(const *Warning* &warning)>
warning callback type; this is passed into `calculate` calls, and will be called with any warnings.

3.8 Changelog

3.8.1 unreleased changes

Changed

- `Layout::screen` defaults to `getDefaultScreen()` to match the EAR. Call `layout.screen(boost::none)` to get the old behaviour.
- added xsimd submodule and updated eigen to 3.4.0; this required changing the eigen remote, so you may need to run `git submodule sync` as well as the usual `git submodule update --init --recursive`

3.8.2 0.9.0

Initial release.

3.9 Bibliography

BIBLIOGRAPHY

[bs2127] Recommendation ITU-R BS.2127-0 : Audio Definition Model renderer for advanced sound systems. URL: https://www.itu.int/rec/R-REC-BS.2127.

[bs2051] Recommendation ITU-R BS.2051-2 : Advanced sound system for programme production. URL: https://www.itu.int/rec/R-REC-BS.2051.

INDEX

E