iir1

Generated on Fri Jul 4 2025 20:15:51 for iir1 by Doxygen 1.9.8

Fri Jul 4 2025 20:15:51

1 DSP IIR Realtime C++ filter library

An infinite impulse response (IIR) filter library for Linux, Mac OSX and Windows which implements Butterworth, RBJ, Chebychev filters and can easily import coefficients generated by Python (scipy).

The filter processes the data sample by sample for realtime processing.

It uses templates to allocate the required memory so that it can run without any malloc / new commands. Memory is allocated at compile time so that there is never any risk of memory leaks.

This library has been further developed from Vinnie Falco's great original work which can be found here:

`https://github.com/vinniefalco/DSPFilters`

Bernd Porr – `http://www.berndporr.me.uk`

2 Namespace Index

2.1 Namespace List

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

Iir	**8**
Iir::Butterworth	**11**
Iir::ChebyshevI	**11**
Iir::ChebyshevII	**12**
Iir::Custom	**12**

3 Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

BandPassBase

4 Class Index

4.1 Class List

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

5 File Index

5.1 File List

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

6 Namespace Documentation

6.1 Iir Namespace Reference

Namespaces

- namespace Butterworth
- namespace ChebyshevI
- namespace ChebyshevII
- namespace Custom

Classes

- class BandPassTransform
- class BandStopTransform
- class Biquad
- struct BiquadPoleState
- class Cascade
- class CascadeStages
- struct ComplexPair
- class DirectFormI
- class DirectFormII
- class HighPassTransform
- class Layout
- class LayoutBase
- class LowPassTransform
- struct PoleFilter
- class PoleFilterBase
- class PoleFilterBase2
- struct PoleZeroPair
- class TransposedDirectFormII

6.1.1 Detailed Description

"A Collection of Useful C++ Classes for Digital Signal Processing" By Vinnie Falco and Bernd Porr

Official project location: `https://github.com/berndporr/iir1`

See Documentation.cpp for contact information, notes, and bibliography.

"A Collection of Useful C++ Classes for Digital Signal Processing" By Vinnie Falco and Bernd Porr

Official project location: `https://github.com/berndporr/iir1`

See Documentation.txt for contact information, notes, and bibliography.

"A Collection of Useful C++ Classes for Digital Signal Processing" By Vinnie Falco and Bernd Porr

Official project location: `https://github.com/berndporr/iir1`

See Documentation.cpp for contact information, notes, and bibliography.

"A Collection of Useful C++ Classes for Digital Signal Processing" By Vinnie Falco and Bernd Porr

Official project location: `https://github.com/berndporr/iir1`

See Documentation.txt for contact information, notes, and bibliography.

"A Collection of Useful C++ Classes for Digital Signal Processing" By Vinnie Falco and Bernd Porr

Official project location: `https://github.com/berndporr/iir1`

See Documentation.cpp for contact information, notes, and bibliography.

Describes a filter as a collection of poles and zeros along with normalization information to achieve a specified gain at a specified frequency. The poles and zeros may lie either in the s or the z plane.

6.2 Iir::Butterworth Namespace Reference

Classes

- class AnalogLowPass
- class AnalogLowShelf
- struct BandPass
- struct BandPassBase
- struct BandShelf
- struct BandShelfBase
- struct BandStop
- struct BandStopBase
- struct HighPass
- struct HighPassBase
- struct HighShelf
- struct HighShelfBase
- struct LowPass
- struct LowPassBase
- struct LowShelf
- struct LowShelfBase

6.2.1 Detailed Description

Filters with Butterworth response characteristics. The filter order is usually set via the template parameter which reserves the correct space and is then automatically passed to the setup function. Optionally one can also provde the filter order at setup time to force a lower order than the default one.

6.3 Iir::ChebyshevI Namespace Reference

Classes

- class AnalogLowPass
- class AnalogLowShelf
- struct BandPass
- struct BandPassBase
- struct BandShelf
- struct BandShelfBase
- struct BandStop
- struct BandStopBase
- struct HighPass
- struct HighPassBase
- struct HighShelf
- struct HighShelfBase
- struct LowPass
- struct LowPassBase
- struct LowShelf
- struct LowShelfBase

6.3.1 Detailed Description

Filters with Chebyshev response characteristics. The last parameter defines the passband ripple in decibel.

6.4 Iir::ChebyshevII Namespace Reference

Classes

- class AnalogLowPass
- class AnalogLowShelf
- struct BandPass
- struct BandPassBase
- struct BandShelf
- struct BandShelfBase
- struct BandStop
- struct BandStopBase
- struct HighPass
- struct HighPassBase
- struct HighShelf
- struct HighShelfBase
- struct LowPass
- struct LowPassBase
- struct LowShelf
- struct LowShelfBase

6.4.1 Detailed Description

Filters with ChebyshevII response characteristics. The last parameter defines the minimal stopband rejection requested. Generally there will be frequencies where the rejection is much better but this parameter guarantees that the rejection is at least as specified.

6.5 Iir::Custom Namespace Reference

Classes

- struct OnePole
- struct SOSCascade
- struct TwoPole

6.5.1 Detailed Description

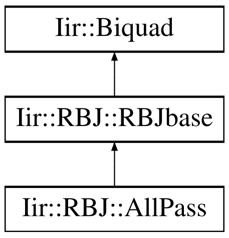
Single pole, Biquad and cascade of Biquads with parameters allowing for directly setting the parameters.

7 Class Documentation

7.1 Iir::RBJ::AllPass Struct Reference

`#include <RBJ.h>`

Inheritance diagram for Iir::RBJ::AllPass:



Public Member Functions

- void setupN (double phaseFrequency, double q=(1/sqrt(2)))
- void setup (double sampleRate, double phaseFrequency, double q=(1/sqrt(2)))

Public Member Functions inherited from Iir::RBJ::RBJbase

- template<typename Sample >
  Sample **filter** (Sample s)

  filter operation
- void **reset** ()

  resets the delay lines to zero
- const DirectFormI & **getState** ()

  gets the delay lines (=state) of the filter

Public Member Functions inherited from Iir::Biquad

- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const
- double getA0 () const
- double getA1 () const
- double getA2 () const
- double getB0 () const
- double getB1 () const
- double getB2 () const
- template<class StateType >
  double filter (double s, StateType &state) const
- void setCoefficients (double a0, double a1, double a2, double b0, double b1, double b2)
- void setOnePole (complex_t pole, complex_t zero)
- void setTwoPole (complex_t pole1, complex_t zero1, complex_t pole2, complex_t zero2)
- void setPoleZeroPair (const PoleZeroPair &pair)
- void setIdentity ()
- void applyScale (double scale)

7.1.1 Detailed Description

Allpass filter

7.1.2 Member Function Documentation

setup()

```
void Iir::RBJ::AllPass::setup (
             double sampleRate,
             double phaseFrequency,
             double q = (1/sqrt(2)) )  [inline]
```

Calculates the coefficients

Parameters

sampleRate	Sampling rate
phaseFrequency	Frequency where the phase flips
q	Q-factor

setupN()

```
void Iir::RBJ::AllPass::setupN (
             double phaseFrequency,
             double q = (1/sqrt(2)) )
```

Calculates the coefficients

Parameters

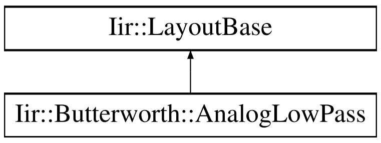
phaseFrequency	Normalised frequency where the phase flips
q	Q-factor

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

- iir/RBJ.h
- iir/RBJ.cpp

7.2 Iir::Butterworth::AnalogLowPass Class Reference

`#include <Butterworth.h>`
Inheritance diagram for Iir::Butterworth::AnalogLowPass:



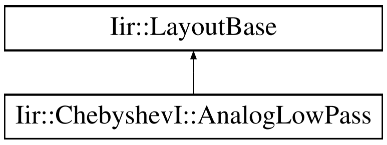
7.2.1 Detailed Description

Analogue lowpass prototypes (s-plane)
The documentation for this class was generated from the following files:

- iir/Butterworth.h
- iir/Butterworth.cpp

7.3 Iir::ChebyshevI::AnalogLowPass Class Reference

`#include <ChebyshevI.h>`
Inheritance diagram for Iir::ChebyshevI::AnalogLowPass:



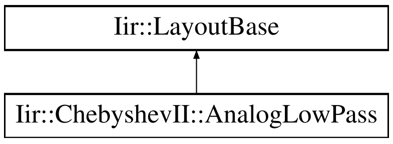
7.3.1 Detailed Description

Analog lowpass prototypes (s-plane)
The documentation for this class was generated from the following files:

- iir/ChebyshevI.h
- iir/ChebyshevI.cpp

7.4 Iir::ChebyshevII::AnalogLowPass Class Reference

`#include <ChebyshevII.h>`
Inheritance diagram for Iir::ChebyshevII::AnalogLowPass:



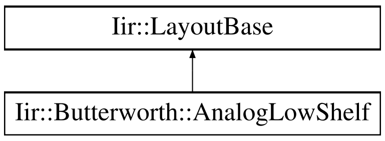
7.4.1 Detailed Description

Analogue lowpass prototype (s-plane)
The documentation for this class was generated from the following files:

- iir/ChebyshevII.h
- iir/ChebyshevII.cpp

7.5 Iir::Butterworth::AnalogLowShelf Class Reference

`#include <Butterworth.h>`
Inheritance diagram for Iir::Butterworth::AnalogLowShelf:



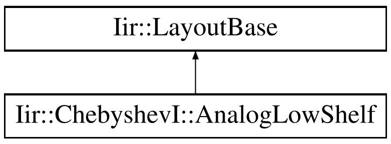
7.5.1 Detailed Description

Analogue low shelf prototypes (s-plane)
The documentation for this class was generated from the following files:

- iir/Butterworth.h
- iir/Butterworth.cpp

7.6 Iir::ChebyshevI::AnalogLowShelf Class Reference

`#include <ChebyshevI.h>`
Inheritance diagram for Iir::ChebyshevI::AnalogLowShelf:



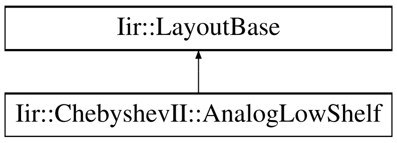
7.6.1 Detailed Description

Analog lowpass shelf prototype (s-plane)
The documentation for this class was generated from the following files:

- iir/ChebyshevI.h
- iir/ChebyshevI.cpp

7.7 Iir::ChebyshevII::AnalogLowShelf Class Reference

`#include <ChebyshevII.h>`
Inheritance diagram for Iir::ChebyshevII::AnalogLowShelf:



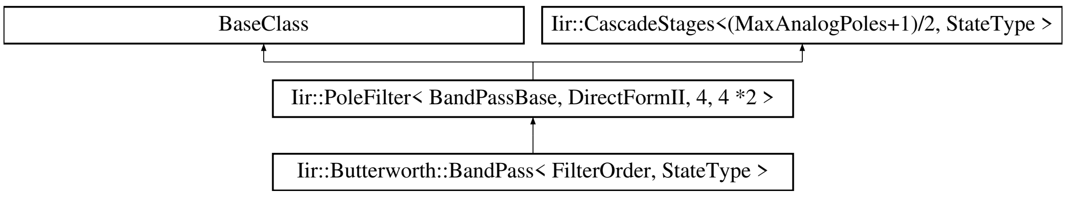
7.7.1 Detailed Description

Analogue shelf lowpass prototype (s-plane)
The documentation for this class was generated from the following files:

- iir/ChebyshevII.h
- iir/ChebyshevII.cpp

7.8 Iir::Butterworth::BandPass< FilterOrder, StateType > Struct Template Reference

`#include <Butterworth.h>`
Inheritance diagram for Iir::Butterworth::BandPass< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< BandPassBase, DirectFormII, 4, 4 *2 >
Iir::Butterworth::BandPass< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double centerFrequency, double widthFrequency)
- void setup (int reqOrder, double sampleRate, double centerFrequency, double widthFrequency)
- void setupN (double centerFrequency, double widthFrequency)
- void setupN (int reqOrder, double centerFrequency, double widthFrequency)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.8.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::Butterworth::BandPass< FilterOrder, StateType >

Butterworth Bandpass filter.

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.8.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::BandPass< FilterOrder, StateType >::setup (
            double sampleRate,
            double centerFrequency,
            double widthFrequency ) [inline]
```

Calculates the coefficients with the filter order provided by the instantiation

Parameters

sampleRate	Sampling rate
centerFrequency	Centre frequency of the bandpass
widthFrequency	Width of the bandpass

setup() `[2/2]`

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::BandPass< FilterOrder, StateType >::setup (
             int reqOrder,
             double sampleRate,
             double centerFrequency,
             double widthFrequency )  [inline]
```

Calculates the coefficients

Parameters

reqOrder	The actual order which can be less than the instantiated one
sampleRate	Sampling rate
centerFrequency	Centre frequency of the bandpass
widthFrequency	Width of the bandpass

setupN() `[1/2]`

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::BandPass< FilterOrder, StateType >::setupN (
             double centerFrequency,
             double widthFrequency )  [inline]
```

Calculates the coefficients with the filter order provided by the instantiation

Parameters

centerFrequency	Normalised centre frequency (0..1/2) of the bandpass
widthFrequency	Width of the bandpass in normalised freq

setupN() `[2/2]`

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::BandPass< FilterOrder, StateType >::setupN (
             int reqOrder,
             double centerFrequency,
             double widthFrequency )  [inline]
```

Calculates the coefficients

Parameters

reqOrder	The actual order which can be less than the instantiated one
centerFrequency	Normalised centre frequency (0..1/2) of the bandpass
widthFrequency	Width of the bandpass in normalised freq

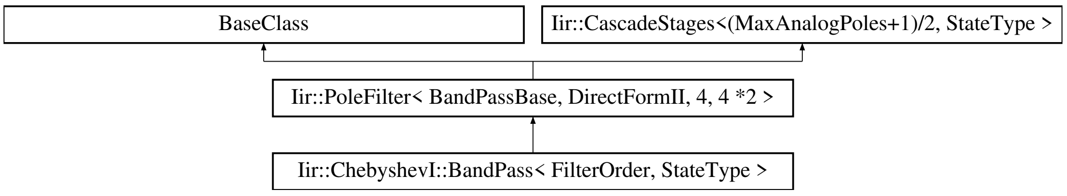
The documentation for this struct was generated from the following file:

- iir/Butterworth.h

7.9 Iir::ChebyshevI::BandPass< FilterOrder, StateType > Struct Template Reference

```
#include <ChebyshevI.h>
```

Inheritance diagram for Iir::ChebyshevI::BandPass< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< BandPassBase, DirectFormII, 4, 4 *2 >
Iir::ChebyshevI::BandPass< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double centerFrequency, double widthFrequency, double rippleDb)
- void setup (int reqOrder, double sampleRate, double centerFrequency, double widthFrequency, double rippleDb)
- void setupN (double centerFrequency, double widthFrequency, double rippleDb)
- void setupN (int reqOrder, double centerFrequency, double widthFrequency, double rippleDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.9.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::ChebyshevI::BandPass< FilterOrder, StateType >

ChebyshevI bandpass filter

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.9.2 Member Function Documentation

setup() `[1/2]`

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::BandPass< FilterOrder, StateType >::setup (
             double sampleRate,
             double centerFrequency,
             double widthFrequency,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at the order FilterOrder

Parameters

sampleRate	Sampling rate
centerFrequency	Center frequency of the bandpass
widthFrequency	Frequency with of the passband

Parameters

rippleDb	Permitted ripples in dB in the passband

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::BandPass< FilterOrder, StateType >::setup (
             int reqOrder,
             double sampleRate,
             double centerFrequency,
             double widthFrequency,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at specified order

Parameters

reqOrder	Actual order for the filter calculations
sampleRate	Sampling rate
centerFrequency	Center frequency of the bandpass
widthFrequency	Frequency with of the passband
rippleDb	Permitted ripples in dB in the passband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::BandPass< FilterOrder, StateType >::setupN (
             double centerFrequency,
             double widthFrequency,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at the order FilterOrder

Parameters

centerFrequency	Normalised center frequency (0..1/2) of the bandpass
widthFrequency	Frequency with of the passband
rippleDb	Permitted ripples in dB in the passband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::BandPass< FilterOrder, StateType >::setupN (
             int reqOrder,
             double centerFrequency,
             double widthFrequency,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at specified order

Parameters

reqOrder	Actual order for the filter calculations
centerFrequency	Normalised center frequency (0..1/2) of the bandpass
widthFrequency	Frequency with of the passband
rippleDb	Permitted ripples in dB in the passband

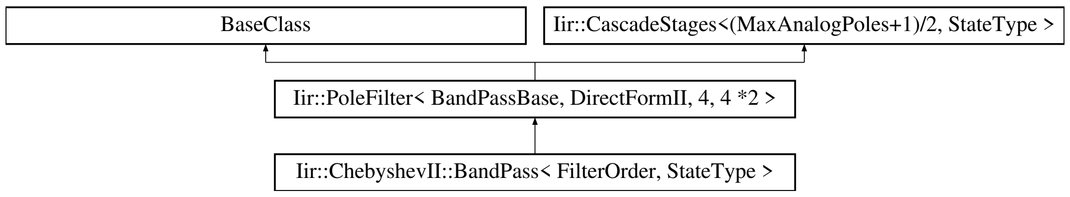
The documentation for this struct was generated from the following file:

- iir/ChebyshevI.h

7.10 Iir::ChebyshevII::BandPass< FilterOrder, StateType > Struct Template Reference

`#include <ChebyshevII.h>`

Inheritance diagram for Iir::ChebyshevII::BandPass< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< BandPassBase, DirectFormII, 4, 4 *2 >
Iir::ChebyshevII::BandPass< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double centerFrequency, double widthFrequency, double stopBandDb)
- void setup (int reqOrder, double sampleRate, double centerFrequency, double widthFrequency, double stop←BandDb)
- void setupN (double centerFrequency, double widthFrequency, double stopBandDb)
- void setupN (int reqOrder, double centerFrequency, double widthFrequency, double stopBandDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.10.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::ChebyshevII::BandPass< FilterOrder, StateType >

ChebyshevII bandpass filter

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.10.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::BandPass< FilterOrder, StateType >::setup (
            double sampleRate,
            double centerFrequency,
            double widthFrequency,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

sampleRate	Sampling rate
centerFrequency	Center frequency of the bandpass
widthFrequency	Width of the bandpass
stopBandDb	Permitted ripples in dB in the stopband

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::BandPass< FilterOrder, StateType >::setup (
            int reqOrder,
            double sampleRate,
            double centerFrequency,
            double widthFrequency,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

reqOrder	Requested order which can be less than the instantiated one
sampleRate	Sampling rate
centerFrequency	Center frequency of the bandpass
widthFrequency	Width of the bandpass
stopBandDb	Permitted ripples in dB in the stopband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::BandPass< FilterOrder, StateType >::setupN (
            double centerFrequency,
            double widthFrequency,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

centerFrequency	Normalised centre frequency (0..1/2) of the bandpass
widthFrequency	Width of the bandpass
stopBandDb	Permitted ripples in dB in the stopband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::BandPass< FilterOrder, StateType >::setupN (
            int reqOrder,
            double centerFrequency,
            double widthFrequency,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

reqOrder	Requested order which can be less than the instantiated one

Parameters

centerFrequency	Normalised centre frequency (0..1/2) of the bandpass
widthFrequency	Width of the bandpass
stopBandDb	Permitted ripples in dB in the stopband

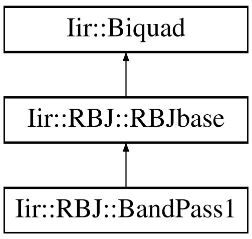
The documentation for this struct was generated from the following file:

- iir/ChebyshevII.h

7.11 Iir::RBJ::BandPass1 Struct Reference

`#include <RBJ.h>`

Inheritance diagram for Iir::RBJ::BandPass1:



Public Member Functions

- void setupN (double centerFrequency, double bandWidth)
- void setup (double sampleRate, double centerFrequency, double bandWidth)

Public Member Functions inherited from Iir::RBJ::RBJbase

- template<typename Sample >
  Sample **filter** (Sample s)

  filter operation
- void **reset** ()

  resets the delay lines to zero
- const DirectFormI & **getState** ()

  gets the delay lines (=state) of the filter

Public Member Functions inherited from Iir::Biquad

- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const
- double getA0 () const
- double getA1 () const
- double getA2 () const
- double getB0 () const
- double getB1 () const
- double getB2 () const
- template<class StateType >
  double filter (double s, StateType &state) const
- void setCoefficients (double a0, double a1, double a2, double b0, double b1, double b2)
- void setOnePole (complex_t pole, complex_t zero)
- void setTwoPole (complex_t pole1, complex_t zero1, complex_t pole2, complex_t zero2)
- void setPoleZeroPair (const PoleZeroPair &pair)
- void setIdentity ()
- void applyScale (double scale)

7.11.1 Detailed Description

Bandpass with constant skirt gain

7.11.2 Member Function Documentation

setup()

```
void Iir::RBJ::BandPass1::setup (
             double sampleRate,
             double centerFrequency,
             double bandWidth )  [inline]
```

Calculates the coefficients

Parameters

sampleRate	Sampling rate
centerFrequency	Center frequency of the bandpass
bandWidth	Bandwidth in octaves

setupN()

```
void Iir::RBJ::BandPass1::setupN (
             double centerFrequency,
             double bandWidth )
```

Calculates the coefficients

Parameters

centerFrequency	Center frequency of the bandpass
bandWidth	Bandwidth in octaves

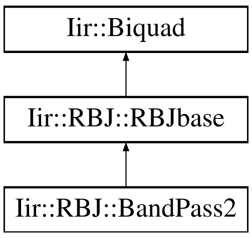
The documentation for this struct was generated from the following files:

- iir/RBJ.h
- iir/RBJ.cpp

7.12 Iir::RBJ::BandPass2 Struct Reference

`#include <RBJ.h>`

Inheritance diagram for Iir::RBJ::BandPass2:



Public Member Functions

- void setupN (double centerFrequency, double bandWidth)
- void setup (double sampleRate, double centerFrequency, double bandWidth)

Public Member Functions inherited from Iir::RBJ::RBJbase

- template<typename Sample >
  Sample **filter** (Sample s)

  filter operation
- void **reset** ()

  resets the delay lines to zero
- const DirectFormI & **getState** ()

  gets the delay lines (=state) of the filter

Public Member Functions inherited from Iir::Biquad

- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const
- double getA0 () const
- double getA1 () const
- double getA2 () const
- double getB0 () const
- double getB1 () const
- double getB2 () const
- template<class StateType >
  double filter (double s, StateType &state) const
- void setCoefficients (double a0, double a1, double a2, double b0, double b1, double b2)
- void setOnePole (complex_t pole, complex_t zero)
- void setTwoPole (complex_t pole1, complex_t zero1, complex_t pole2, complex_t zero2)
- void setPoleZeroPair (const PoleZeroPair &pair)
- void setIdentity ()
- void applyScale (double scale)

7.12.1 Detailed Description

Bandpass with constant 0 dB peak gain

7.12.2 Member Function Documentation

setup()

```
void Iir::RBJ::BandPass2::setup (
             double sampleRate,
             double centerFrequency,
             double bandWidth )  [inline]
```

Calculates the coefficients

Parameters

sampleRate	Sampling rate
centerFrequency	Center frequency of the bandpass
bandWidth	Bandwidth in octaves

setupN()

```
void Iir::RBJ::BandPass2::setupN (
             double centerFrequency,
             double bandWidth )
```

Calculates the coefficients

Parameters

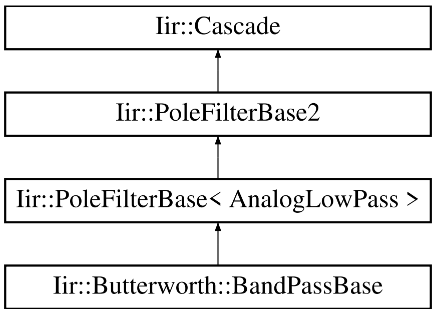
centerFrequency	Normalised centre frequency of the bandpass
bandWidth	Bandwidth in octaves

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

- iir/RBJ.h
- iir/RBJ.cpp

7.13 Iir::Butterworth::BandPassBase Struct Reference

Inheritance diagram for Iir::Butterworth::BandPassBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowPass >
Iir::Butterworth::BandPassBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

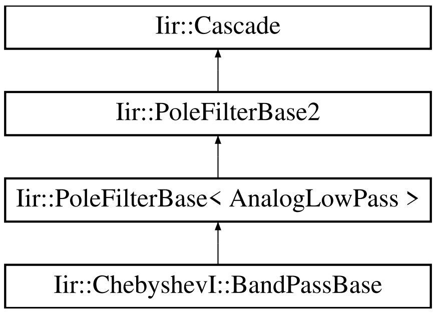
- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/Butterworth.h
- iir/Butterworth.cpp

7.14 Iir::ChebyshevI::BandPassBase Struct Reference

Inheritance diagram for Iir::ChebyshevI::BandPassBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowPass >
Iir::ChebyshevI::BandPassBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

- int getNumStages () const

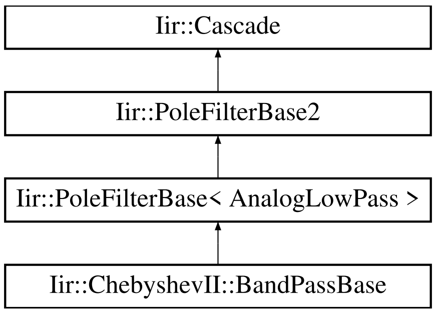
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/ChebyshevI.h
- iir/ChebyshevI.cpp

7.15 Iir::ChebyshevII::BandPassBase Struct Reference

Inheritance diagram for Iir::ChebyshevII::BandPassBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowPass >
Iir::ChebyshevII::BandPassBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/ChebyshevII.h
- iir/ChebyshevII.cpp

7.16 Iir::BandPassTransform Class Reference

```
#include <PoleFilter.h>
```

7.16.1 Detailed Description

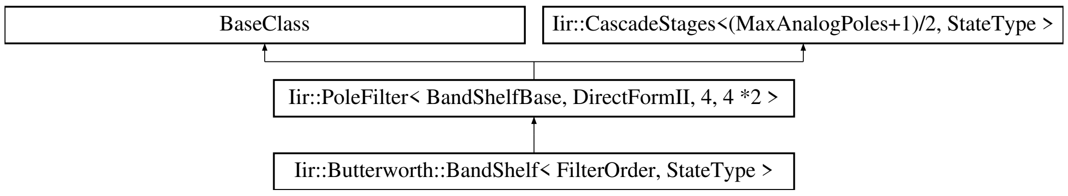
low pass to band pass transform
The documentation for this class was generated from the following files:

- iir/PoleFilter.h
- iir/PoleFilter.cpp

7.17 Iir::Butterworth::BandShelf< FilterOrder, StateType > Struct Template Reference

```
#include <Butterworth.h>
```

Inheritance diagram for Iir::Butterworth::BandShelf< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< BandShelfBase, DirectFormII, 4, 4 *2 >
Iir::Butterworth::BandShelf< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double centerFrequency, double widthFrequency, double gainDb)
- void setup (int reqOrder, double sampleRate, double centerFrequency, double widthFrequency, double gain↩ Db)
- void setupN (double centerFrequency, double widthFrequency, double gainDb)
- void setupN (int reqOrder, double centerFrequency, double widthFrequency, double gainDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.17.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::Butterworth::BandShelf< FilterOrder, StateType >

Butterworth Bandshelf filter: it is a bandpass filter which amplifies at a specified gain in dB the frequencies in the passband.

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.17.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::BandShelf< FilterOrder, StateType >::setup (
            double sampleRate,
            double centerFrequency,
            double widthFrequency,
            double gainDb )  [inline]
```

Calculates the coefficients with the filter order provided by the instantiation

Parameters

sampleRate	Sampling rate
centerFrequency	Centre frequency of the passband
widthFrequency	Width of the passband
gainDb	The gain in the passband

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::BandShelf< FilterOrder, StateType >::setup (
            int reqOrder,
            double sampleRate,
            double centerFrequency,
            double widthFrequency,
            double gainDb )  [inline]
```

Calculates the coefficients

Parameters

reqOrder	The actual order which can be less than the instantiated one
sampleRate	Sampling rate
centerFrequency	Centre frequency of the passband
widthFrequency	Width of the passband
gainDb	The gain in the passband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::BandShelf< FilterOrder, StateType >::setupN (
            double centerFrequency,
            double widthFrequency,
            double gainDb )  [inline]
```

Calculates the coefficients with the filter order provided by the instantiation

Parameters

centerFrequency	Normalised centre frequency (0..1/2) of the passband
widthFrequency	Width of the passband
gainDb	The gain in the passband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::BandShelf< FilterOrder, StateType >::setupN (
            int reqOrder,
            double centerFrequency,
            double widthFrequency,
            double gainDb )  [inline]
```

Calculates the coefficients

Parameters

reqOrder	The actual order which can be less than the instantiated one
centerFrequency	Normalised centre frequency (0..1/2) of the passband
widthFrequency	Width of the passband
gainDb	The gain in the passband

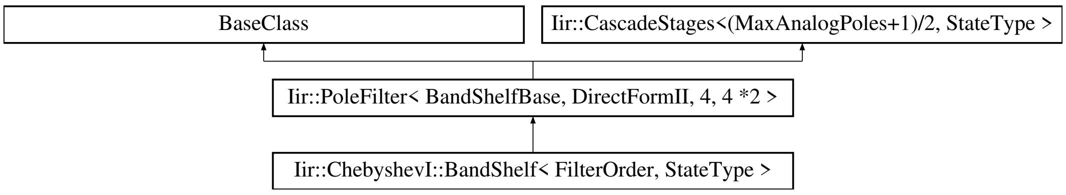
The documentation for this struct was generated from the following file:

- iir/Butterworth.h

7.18 Iir::ChebyshevI::BandShelf< FilterOrder, StateType > Struct Template Reference

```
#include <ChebyshevI.h>
```

Inheritance diagram for Iir::ChebyshevI::BandShelf< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< BandShelfBase, DirectFormII, 4, 4 *2 >
Iir::ChebyshevI::BandShelf< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double centerFrequency, double widthFrequency, double gainDb, double rippleDb)
- void setup (int reqOrder, double sampleRate, double centerFrequency, double widthFrequency, double gain↩ Db, double rippleDb)
- void setupN (double centerFrequency, double widthFrequency, double gainDb, double rippleDb)
- void setupN (int reqOrder, double centerFrequency, double widthFrequency, double gainDb, double rippleDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.18.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::ChebyshevI::BandShelf< FilterOrder, StateType >

ChebyshevI bandshelf filter. Specified gain in the passband. Otherwise 0 dB.

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.18.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::BandShelf< FilterOrder, StateType >::setup (
            double sampleRate,
            double centerFrequency,
            double widthFrequency,
            double gainDb,
            double rippleDb )  [inline]
```

Calculates the coefficients of the filter at the order FilterOrder

Parameters

sampleRate	Sampling rate
centerFrequency	Center frequency of the passband
widthFrequency	Width of the passband.
gainDb	Gain in the passband. The stopband has 0 dB.
rippleDb	Permitted ripples in dB in the passband.

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::BandShelf< FilterOrder, StateType >::setup (
             int reqOrder,
             double sampleRate,
             double centerFrequency,
             double widthFrequency,
             double gainDb,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at specified order

Parameters

reqOrder	Actual order for the filter calculations
sampleRate	Sampling rate
centerFrequency	Center frequency of the passband
widthFrequency	Width of the passband.
gainDb	Gain in the passband. The stopband has 0 dB.
rippleDb	Permitted ripples in dB in the passband.

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::BandShelf< FilterOrder, StateType >::setupN (
             double centerFrequency,
             double widthFrequency,
             double gainDb,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at the order FilterOrder

Parameters

centerFrequency	Normalised centre frequency (0..1/2) of the passband
widthFrequency	Width of the passband.
gainDb	Gain in the passband. The stopband has 0 dB.
rippleDb	Permitted ripples in dB in the passband.

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::BandShelf< FilterOrder, StateType >::setupN (
             int reqOrder,
             double centerFrequency,
             double widthFrequency,
             double gainDb,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at specified order

Parameters

reqOrder	Actual order for the filter calculations
centerFrequency	Normalised centre frequency (0..1/2) of the passband
widthFrequency	Width of the passband.
gainDb	Gain in the passband. The stopband has 0 dB.

Parameters

rippleDb	Permitted ripples in dB in the passband.

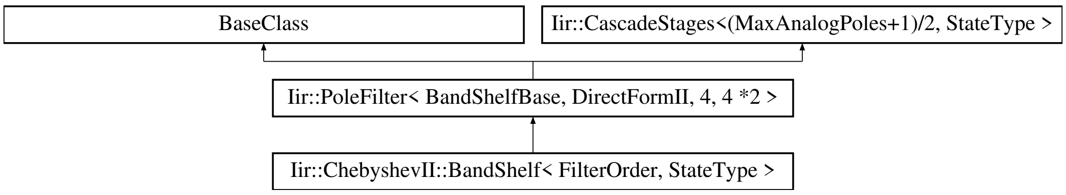
The documentation for this struct was generated from the following file:

- iir/ChebyshevI.h

7.19 Iir::ChebyshevII::BandShelf< FilterOrder, StateType > Struct Template Reference

`#include <ChebyshevII.h>`

Inheritance diagram for Iir::ChebyshevII::BandShelf< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< BandShelfBase, DirectFormII, 4, 4 *2 >
Iir::ChebyshevII::BandShelf< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double centerFrequency, double widthFrequency, double gainDb, double stopBandDb)
- void setup (int reqOrder, double sampleRate, double centerFrequency, double widthFrequency, double gain↩ Db, double stopBandDb)
- void setupN (double centerFrequency, double widthFrequency, double gainDb, double stopBandDb)
- void setupN (int reqOrder, double centerFrequency, double widthFrequency, double gainDb, double stop↩ BandDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.19.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::ChebyshevII::BandShelf< FilterOrder, StateType >

ChebyshevII bandshelf filter. Bandpass with specified gain and 0 dB gain in the stopband.

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.19.2 Member Function Documentation

setup() `[1/2]`

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::BandShelf< FilterOrder, StateType >::setup (
            double sampleRate,
            double centerFrequency,
            double widthFrequency,
            double gainDb,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

sampleRate	Sampling rate
centerFrequency	Center frequency of the bandpass
widthFrequency	Width of the bandpass
gainDb	Gain in the passband. The stopband has always 0dB.
stopBandDb	Permitted ripples in dB in the stopband

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::BandShelf< FilterOrder, StateType >::setup (
             int reqOrder,
             double sampleRate,
             double centerFrequency,
             double widthFrequency,
             double gainDb,
             double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

reqOrder	Requested order which can be less than the instantiated one
sampleRate	Sampling rate
centerFrequency	Center frequency of the bandpass
widthFrequency	Width of the bandpass
gainDb	Gain in the passband. The stopband has always 0dB.
stopBandDb	Permitted ripples in dB in the stopband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::BandShelf< FilterOrder, StateType >::setupN (
             double centerFrequency,
             double widthFrequency,
             double gainDb,
             double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

centerFrequency	Normalised centre frequency (0..1/2) of the bandpass
widthFrequency	Width of the bandpass
gainDb	Gain in the passband. The stopband has always 0dB.
stopBandDb	Permitted ripples in dB in the stopband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::BandShelf< FilterOrder, StateType >::setupN (
             int reqOrder,
             double centerFrequency,
             double widthFrequency,
             double gainDb,
             double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

reqOrder	Requested order which can be less than the instantiated one
centerFrequency	Normalised centre frequency (0..1/2) of the bandpass
widthFrequency	Width of the bandpass
gainDb	Gain in the passband. The stopband has always 0dB.
stopBandDb	Permitted ripples in dB in the stopband

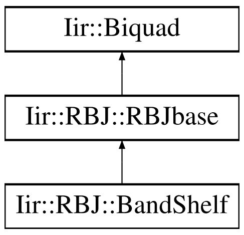
The documentation for this struct was generated from the following file:

- iir/ChebyshevII.h

7.20 Iir::RBJ::BandShelf Struct Reference

`#include <RBJ.h>`

Inheritance diagram for Iir::RBJ::BandShelf:



Public Member Functions

- void setupN (double centerFrequency, double gainDb, double bandWidth)
- void setup (double sampleRate, double centerFrequency, double gainDb, double bandWidth)

Public Member Functions inherited from Iir::RBJ::RBJbase

- template<typename Sample >
  Sample **filter** (Sample s)

  filter operation
- void **reset** ()

  resets the delay lines to zero
- const DirectFormI & **getState** ()

  gets the delay lines (=state) of the filter

Public Member Functions inherited from Iir::Biquad

- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const
- double getA0 () const

- double getA1 () const
- double getA2 () const
- double getB0 () const
- double getB1 () const
- double getB2 () const
- template<class StateType >
  double filter (double s, StateType &state) const
- void setCoefficients (double a0, double a1, double a2, double b0, double b1, double b2)
- void setOnePole (complex_t pole, complex_t zero)
- void setTwoPole (complex_t pole1, complex_t zero1, complex_t pole2, complex_t zero2)
- void setPoleZeroPair (const PoleZeroPair &pair)
- void setIdentity ()
- void applyScale (double scale)

7.20.1 Detailed Description

Band shelf: 0db in the stopband and gainDb in the passband.

7.20.2 Member Function Documentation

setup()

```
void Iir::RBJ::BandShelf::setup (
             double sampleRate,
             double centerFrequency,
             double gainDb,
             double bandWidth )  [inline]
```

Calculates the coefficients

Parameters

sampleRate	Sampling rate
centerFrequency	frequency
gainDb	Gain in the passband
bandWidth	Bandwidth in octaves

setupN()

```
void Iir::RBJ::BandShelf::setupN (
             double centerFrequency,
             double gainDb,
             double bandWidth )
```

Calculates the coefficients

Parameters

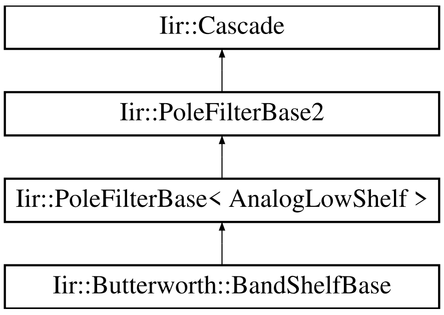
centerFrequency	Normalised centre frequency
gainDb	Gain in the passband
bandWidth	Bandwidth in octaves

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

- iir/RBJ.h
- iir/RBJ.cpp

7.21 Iir::Butterworth::BandShelfBase Struct Reference

Inheritance diagram for Iir::Butterworth::BandShelfBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowShelf >
Iir::Butterworth::BandShelfBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

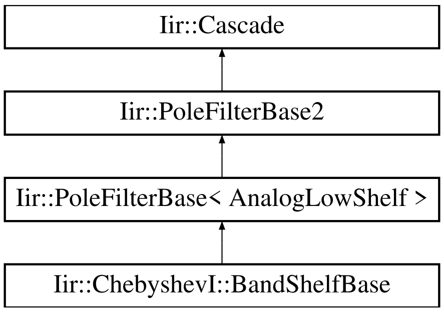
- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/Butterworth.h
- iir/Butterworth.cpp

7.22 Iir::ChebyshevI::BandShelfBase Struct Reference

Inheritance diagram for Iir::ChebyshevI::BandShelfBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowShelf >
Iir::ChebyshevI::BandShelfBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

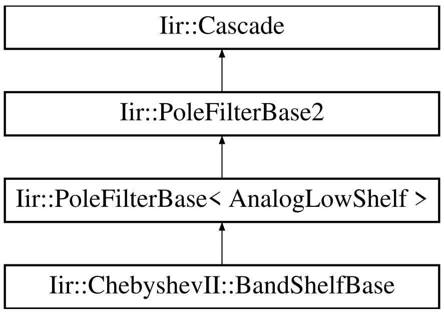
- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/ChebyshevI.h
- iir/ChebyshevI.cpp

7.23 Iir::ChebyshevII::BandShelfBase Struct Reference

Inheritance diagram for Iir::ChebyshevII::BandShelfBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowShelf >
Iir::ChebyshevII::BandShelfBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

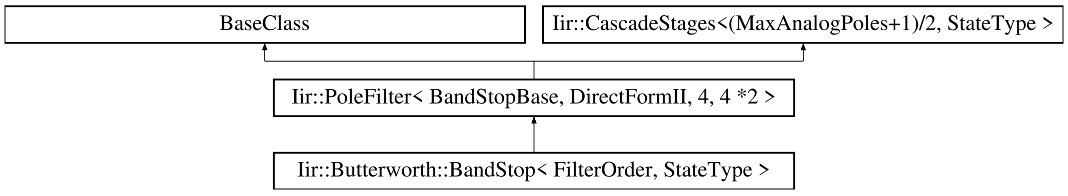
- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/ChebyshevII.h
- iir/ChebyshevII.cpp

7.24 Iir::Butterworth::BandStop< FilterOrder, StateType > Struct Template Reference

```
#include <Butterworth.h>
```
Inheritance diagram for Iir::Butterworth::BandStop< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< BandStopBase, DirectFormII, 4, 4 *2 >
Iir::Butterworth::BandStop< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double centerFrequency, double widthFrequency)
- void setup (int reqOrder, double sampleRate, double centerFrequency, double widthFrequency)
- void setupN (double centerFrequency, double widthFrequency)
- void setupN (int reqOrder, double centerFrequency, double widthFrequency)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.24.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::Butterworth::BandStop< FilterOrder, StateType >

Butterworth Bandstop filter.

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.24.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::BandStop< FilterOrder, StateType >::setup (
             double sampleRate,
             double centerFrequency,
             double widthFrequency )  [inline]
```

Calculates the coefficients with the filter order provided by the instantiation

Parameters

sampleRate	Sampling rate
centerFrequency	Centre frequency of the bandstop
widthFrequency	Width of the bandstop

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::BandStop< FilterOrder, StateType >::setup (
             int reqOrder,
             double sampleRate,
             double centerFrequency,
             double widthFrequency )  [inline]
```

Calculates the coefficients

Parameters

reqOrder	The actual order which can be less than the instantiated one
sampleRate	Sampling rate
centerFrequency	Centre frequency of the bandstop
widthFrequency	Width of the bandstop

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::BandStop< FilterOrder, StateType >::setupN (
             double centerFrequency,
             double widthFrequency )  [inline]
```

Calculates the coefficients with the filter order provided by the instantiation

Parameters

centerFrequency	Normalised centre frequency (0..1/2) of the bandstop
widthFrequency	Normalised width of the bandstop

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::BandStop< FilterOrder, StateType >::setupN (
            int reqOrder,
            double centerFrequency,
            double widthFrequency )  [inline]
```

Calculates the coefficients

Parameters

reqOrder	The actual order which can be less than the instantiated one
centerFrequency	Normalised centre frequency (0..1/2) of the bandstop
widthFrequency	Normalised width of the bandstop

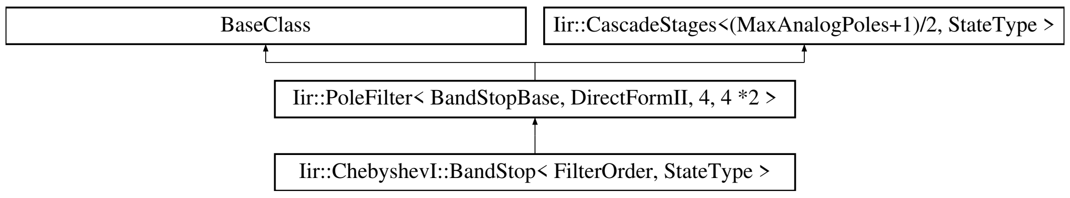
The documentation for this struct was generated from the following file:

- iir/Butterworth.h

7.25 Iir::ChebyshevI::BandStop< FilterOrder, StateType > Struct Template Reference

```
#include <ChebyshevI.h>
```

Inheritance diagram for Iir::ChebyshevI::BandStop< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< BandStopBase, DirectFormII, 4, 4 *2 >
Iir::ChebyshevI::BandStop< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double centerFrequency, double widthFrequency, double rippleDb)
- void setup (int reqOrder, double sampleRate, double centerFrequency, double widthFrequency, double rippleDb)
- void setupN (double centerFrequency, double widthFrequency, double rippleDb)
- void setupN (int reqOrder, double centerFrequency, double widthFrequency, double rippleDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.25.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::ChebyshevI::BandStop< FilterOrder, StateType >

ChebyshevI bandstop filter

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.25.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::BandStop< FilterOrder, StateType >::setup (
             double sampleRate,
             double centerFrequency,
             double widthFrequency,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at the order FilterOrder

Parameters

sampleRate	Sampling rate
centerFrequency	Center frequency of the notch
widthFrequency	Frequency with of the notch
rippleDb	Permitted ripples in dB in the passband

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::BandStop< FilterOrder, StateType >::setup (
             int reqOrder,
             double sampleRate,
             double centerFrequency,
             double widthFrequency,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at specified order

Parameters

reqOrder	Actual order for the filter calculations
sampleRate	Sampling rate
centerFrequency	Center frequency of the notch
widthFrequency	Frequency with of the notch
rippleDb	Permitted ripples in dB in the passband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::BandStop< FilterOrder, StateType >::setupN (
             double centerFrequency,
             double widthFrequency,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at the order FilterOrder

Parameters

centerFrequency	Normalised centre frequency (0..1/2) of the notch
widthFrequency	Frequency width of the notch
rippleDb	Permitted ripples in dB in the passband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::BandStop< FilterOrder, StateType >::setupN (
            int reqOrder,
            double centerFrequency,
            double widthFrequency,
            double rippleDb )  [inline]
```

Calculates the coefficients of the filter at specified order

Parameters

reqOrder	Actual order for the filter calculations
centerFrequency	Normalised centre frequency (0..1/2) of the notch
widthFrequency	Frequency width of the notch
rippleDb	Permitted ripples in dB in the passband

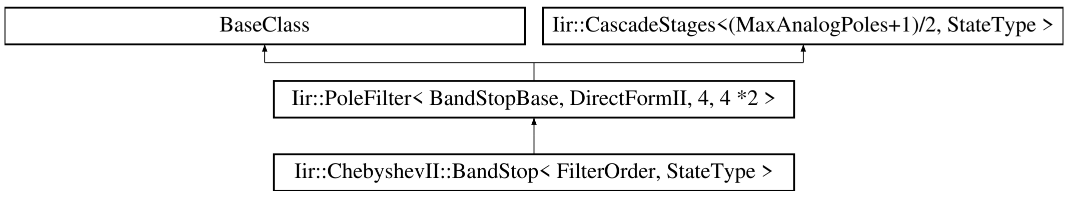
The documentation for this struct was generated from the following file:

- iir/ChebyshevI.h

7.26 Iir::ChebyshevII::BandStop< FilterOrder, StateType > Struct Template Reference

```
#include <ChebyshevII.h>
```

Inheritance diagram for Iir::ChebyshevII::BandStop< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< BandStopBase, DirectFormII, 4, 4 *2 >
Iir::ChebyshevII::BandStop< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double centerFrequency, double widthFrequency, double stopBandDb)
- void setup (int reqOrder, double sampleRate, double centerFrequency, double widthFrequency, double stop↩ BandDb)
- void setupN (double centerFrequency, double widthFrequency, double stopBandDb)
- void setupN (int reqOrder, double centerFrequency, double widthFrequency, double stopBandDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])

- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.26.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::ChebyshevII::BandStop< FilterOrder, StateType >

ChebyshevII bandstop filter.

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.26.2 Member Function Documentation

setup() `[1/2]`

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::BandStop< FilterOrder, StateType >::setup (
            double sampleRate,
            double centerFrequency,
            double widthFrequency,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

sampleRate	Sampling rate
centerFrequency	Center frequency of the bandstop
widthFrequency	Width of the bandstop
stopBandDb	Permitted ripples in dB in the stopband

setup() `[2/2]`

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::BandStop< FilterOrder, StateType >::setup (
            int reqOrder,
            double sampleRate,
            double centerFrequency,
            double widthFrequency,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

reqOrder	Requested order which can be less than the instantiated one
sampleRate	Sampling rate
centerFrequency	Center frequency of the bandstop
widthFrequency	Width of the bandstop
stopBandDb	Permitted ripples in dB in the stopband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::BandStop< FilterOrder, StateType >::setupN (
            double centerFrequency,
            double widthFrequency,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

centerFrequency	Normalised centre frequency (0..1/2) of the bandstop
widthFrequency	Width of the bandstop
stopBandDb	Permitted ripples in dB in the stopband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::BandStop< FilterOrder, StateType >::setupN (
            int reqOrder,
            double centerFrequency,
            double widthFrequency,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

reqOrder	Requested order which can be less than the instantiated one
centerFrequency	Normalised centre frequency (0..1/2) of the bandstop
widthFrequency	Width of the bandstop
stopBandDb	Permitted ripples in dB in the stopband

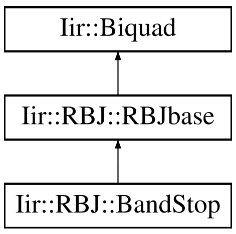
The documentation for this struct was generated from the following file:

- iir/ChebyshevII.h

7.27 Iir::RBJ::BandStop Struct Reference

`#include <RBJ.h>`

Inheritance diagram for Iir::RBJ::BandStop:



Public Member Functions

- void setupN (double centerFrequency, double bandWidth)
- void setup (double sampleRate, double centerFrequency, double bandWidth)

Public Member Functions inherited from Iir::RBJ::RBJbase

- template<typename Sample >
  Sample **filter** (Sample s)

  filter operation
- void **reset** ()

  resets the delay lines to zero
- const DirectFormI & **getState** ()

  gets the delay lines (=state) of the filter

Public Member Functions inherited from Iir::Biquad

- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const
- double getA0 () const
- double getA1 () const
- double getA2 () const
- double getB0 () const
- double getB1 () const
- double getB2 () const
- template<class StateType >
  double filter (double s, StateType &state) const
- void setCoefficients (double a0, double a1, double a2, double b0, double b1, double b2)
- void setOnePole (complex_t pole, complex_t zero)
- void setTwoPole (complex_t pole1, complex_t zero1, complex_t pole2, complex_t zero2)
- void setPoleZeroPair (const PoleZeroPair &pair)
- void setIdentity ()
- void applyScale (double scale)

7.27.1 Detailed Description

Bandstop filter. Warning: the bandwidth might not be accurate for narrow notches.

7.27.2 Member Function Documentation

setup()

```
void Iir::RBJ::BandStop::setup (
            double sampleRate,
            double centerFrequency,
            double bandWidth )  [inline]
```

Calculates the coefficients

Parameters

sampleRate	Sampling rate
centerFrequency	Center frequency of the bandstop
bandWidth	Bandwidth in octaves

setupN()

```
void Iir::RBJ::BandStop::setupN (
            double centerFrequency,
            double bandWidth )
```

Calculates the coefficients

Parameters

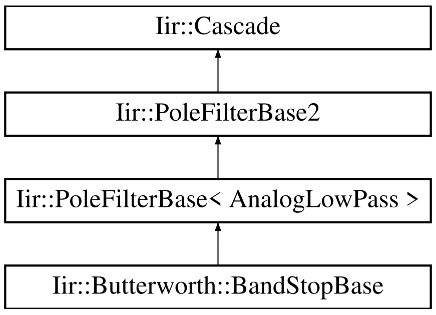
centerFrequency	Normalised Centre frequency of the bandstop
bandWidth	Bandwidth in octaves

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

- iir/RBJ.h
- iir/RBJ.cpp

7.28 Iir::Butterworth::BandStopBase Struct Reference

Inheritance diagram for Iir::Butterworth::BandStopBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowPass >
Iir::Butterworth::BandStopBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

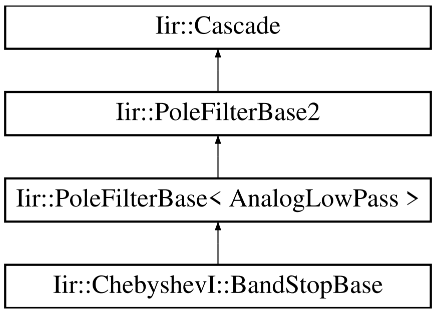
- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/Butterworth.h
- iir/Butterworth.cpp

7.29 Iir::ChebyshevI::BandStopBase Struct Reference

Inheritance diagram for Iir::ChebyshevI::BandStopBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowPass >
Iir::ChebyshevI::BandStopBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

- int getNumStages () const

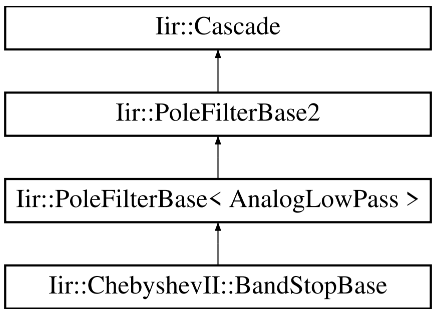
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/ChebyshevI.h
- iir/ChebyshevI.cpp

7.30 Iir::ChebyshevII::BandStopBase Struct Reference

Inheritance diagram for Iir::ChebyshevII::BandStopBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowPass >
Iir::ChebyshevII::BandStopBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/ChebyshevII.h
- iir/ChebyshevII.cpp

7.31 Iir::BandStopTransform Class Reference

```
#include <PoleFilter.h>
```

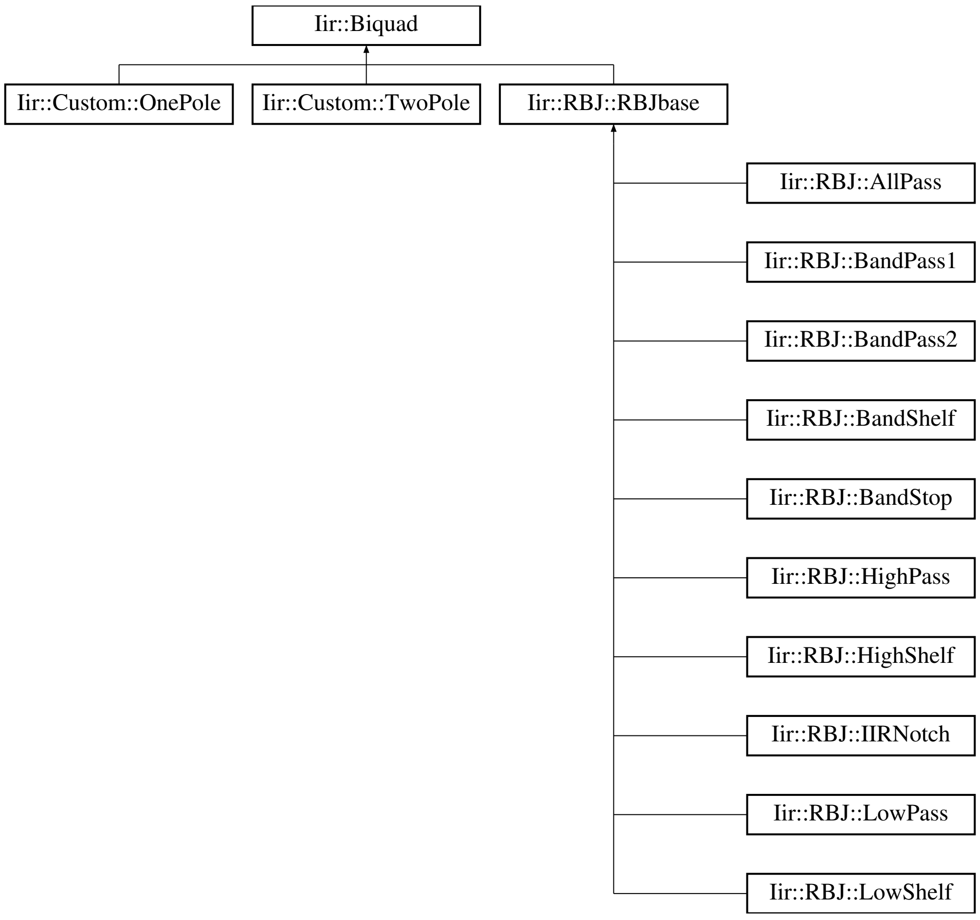
7.31.1 Detailed Description

low pass to band stop transform
The documentation for this class was generated from the following files:

- iir/PoleFilter.h
- iir/PoleFilter.cpp

7.32 Iir::Biquad Class Reference

Inheritance diagram for Iir::Biquad:



Public Member Functions

- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const
- double getA0 () const
- double getA1 () const
- double getA2 () const
- double getB0 () const
- double getB1 () const
- double getB2 () const
- template<class StateType >
  double filter (double s, StateType &state) const
- void setCoefficients (double a0, double a1, double a2, double b0, double b1, double b2)
- void setOnePole (complex_t pole, complex_t zero)
- void setTwoPole (complex_t pole1, complex_t zero1, complex_t pole2, complex_t zero2)
- void setPoleZeroPair (const PoleZeroPair &pair)
- void setIdentity ()
- void applyScale (double scale)

7.32.1 Member Function Documentation

applyScale()

```
void Iir::Biquad::applyScale (
             double scale )
```

Performs scaling operation on the FIR coefficients

Parameters

scale	Mulitplies the coefficients b0,b1,b2 with the scaling factor scale.

filter()

```
template<class StateType >
double Iir::Biquad::filter (
              double s,
              StateType & state ) const  [inline]
```

Filter a sample with the coefficients provided here and the State provided as an argument.

Parameters

s	The sample to be filtered.
state	The Delay lines (instance of a state from State.h)

Returns

The filtered sample.

getA0()

```
double Iir::Biquad::getA0 ( ) const  [inline]
```

Returns 1st IIR coefficient (usually one)

getA1()

```
double Iir::Biquad::getA1 ( ) const  [inline]
```

Returns 2nd IIR coefficient

getA2()

```
double Iir::Biquad::getA2 ( ) const  [inline]
```

Returns 3rd IIR coefficient

getB0()

```
double Iir::Biquad::getB0 ( ) const  [inline]
```

Returns 1st FIR coefficient

getB1()

```
double Iir::Biquad::getB1 ( ) const  [inline]
```

Returns 2nd FIR coefficient

getB2()

```
double Iir::Biquad::getB2 ( ) const  [inline]
```

Returns 3rd FIR coefficient

getPoleZeros()

```
std::vector< PoleZeroPair > Iir::Biquad::getPoleZeros ( ) const
```

Returns the pole / zero Pairs as a vector.

response()

```
complex_t Iir::Biquad::response (
             double normalizedFrequency ) const
```

Calculate filter response at the given normalized frequency and return the complex response.

Gets the frequency response of the Biquad

Parameters

normalizedFrequency	Normalised frequency (0 to 0.5)

setCoefficients()

```
void Iir::Biquad::setCoefficients (
             double a0,
             double a1,
             double a2,
             double b0,
             double b1,
             double b2 )
```

Sets all coefficients

Parameters

a0	1st IIR coefficient
a1	2nd IIR coefficient
a2	3rd IIR coefficient
b0	1st FIR coefficient
b1	2nd FIR coefficient
b2	3rd FIR coefficient

setIdentity()

```
void Iir::Biquad::setIdentity ( )
```

Sets the coefficiens as pass through. (b0=1,a0=1, rest zero)

setOnePole()

```
void Iir::Biquad::setOnePole (
             complex_t pole,
             complex_t zero )
```

Sets one (real) pole and zero. Throws exception if imaginary components.

setPoleZeroPair()

```
void Iir::Biquad::setPoleZeroPair (
             const PoleZeroPair & pair )  [inline]
```

Sets a complex conjugate pair

setTwoPole()

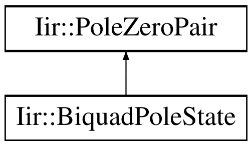
```
void Iir::Biquad::setTwoPole (
             complex_t pole1,
             complex_t zero1,
             complex_t pole2,
             complex_t zero2 )
```

Sets two poles/zoes as a pair. Needs to be complex conjugate.
The documentation for this class was generated from the following files:

- iir/Biquad.h
- iir/Biquad.cpp

7.33 Iir::BiquadPoleState Struct Reference

`#include <Biquad.h>`
Inheritance diagram for Iir::BiquadPoleState:



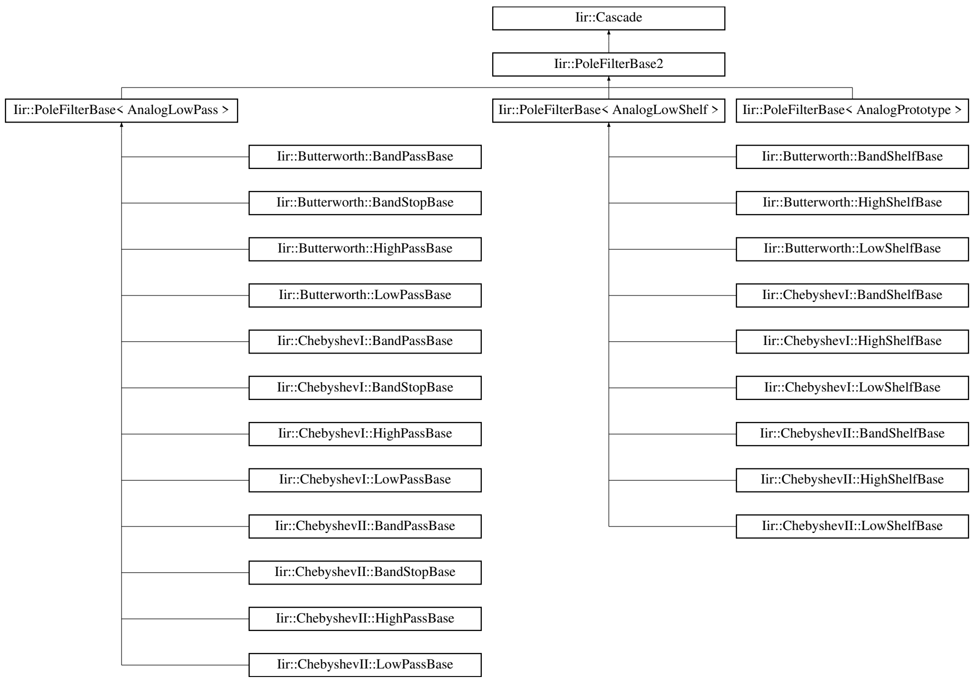
7.33.1 Detailed Description

Expresses a biquad as a pair of pole/zeros, with gain values so that the coefficients can be reconstructed precisely.
The documentation for this struct was generated from the following files:

- iir/Biquad.h
- iir/Biquad.cpp

7.34 Iir::Cascade Class Reference

`#include <Cascade.h>`
Inheritance diagram for Iir::Cascade:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowPass >
Iir::PoleFilterBase< AnalogLowShelf >
Iir::PoleFilterBase< AnalogPrototype >
Iir::Butterworth::BandPassBase
Iir::Butterworth::BandStopBase
Iir::Butterworth::HighPassBase
Iir::Butterworth::LowPassBase
Iir::ChebyshevI::BandPassBase
Iir::ChebyshevI::BandStopBase
Iir::ChebyshevI::HighPassBase
Iir::ChebyshevI::LowPassBase
Iir::ChebyshevII::BandPassBase
Iir::ChebyshevII::BandStopBase
Iir::ChebyshevII::HighPassBase
Iir::ChebyshevII::LowPassBase
Iir::Butterworth::BandShelfBase
Iir::Butterworth::HighShelfBase
Iir::Butterworth::LowShelfBase
Iir::ChebyshevI::BandShelfBase
Iir::ChebyshevI::HighShelfBase
Iir::ChebyshevI::LowShelfBase
Iir::ChebyshevII::BandShelfBase
Iir::ChebyshevII::HighShelfBase
Iir::ChebyshevII::LowShelfBase

Classes

- struct Storage

Public Member Functions

- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

7.34.1 Detailed Description

Holds coefficients for a cascade of second order sections.

7.34.2 Member Function Documentation

getNumStages()

```
int Iir::Cascade::getNumStages ( ) const  [inline]
```
Returns the number of Biquads kept here

getPoleZeros()

```
std::vector< PoleZeroPair > Iir::Cascade::getPoleZeros ( ) const
```
Returns a vector with all pole/zero pairs of the whole Biqad cascade

operator[]()

```
const Biquad & Iir::Cascade::operator[ ] (
             int index )  [inline]
```
Returns a reference to a biquad

response()

```
complex_t Iir::Cascade::response (
             double normalizedFrequency ) const
```
Calculate filter response at the given normalized frequency

Parameters

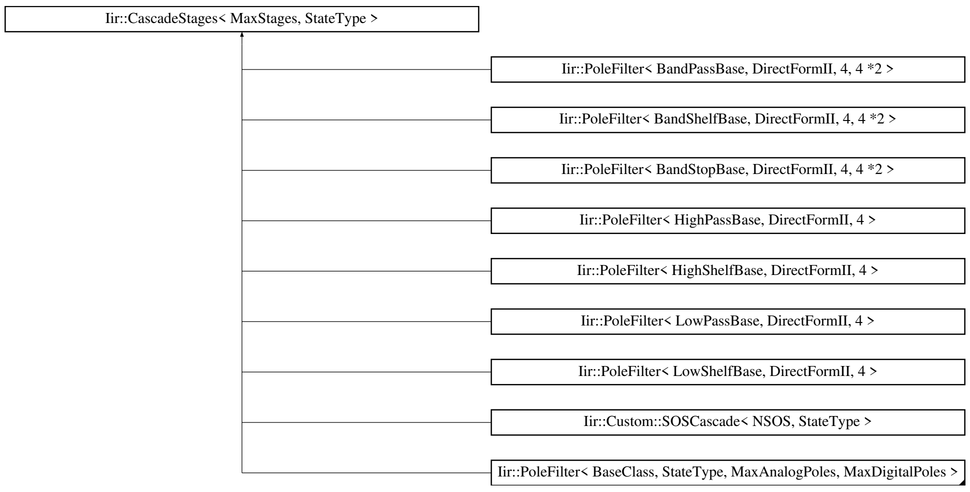
normalizedFrequency	Frequency from 0 to 0.5 (Nyquist)

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

- iir/Cascade.h
- iir/Cascade.cpp

7.35 Iir::CascadeStages< MaxStages, StateType > Class Template Reference

```
#include <Cascade.h>
```
Inheritance diagram for Iir::CascadeStages< MaxStages, StateType >:


Iir::CascadeStages< MaxStages, StateType >
Iir::PoleFilter< BandPassBase, DirectFormII, 4, 4 *2 >
Iir::PoleFilter< BandShelfBase, DirectFormII, 4, 4 *2 >
Iir::PoleFilter< BandStopBase, DirectFormII, 4, 4 *2 >
Iir::PoleFilter< HighPassBase, DirectFormII, 4 >
Iir::PoleFilter< HighShelfBase, DirectFormII, 4 >
Iir::PoleFilter< LowPassBase, DirectFormII, 4 >
Iir::PoleFilter< LowShelfBase, DirectFormII, 4 >
Iir::Custom::SOSCascade< NSOS, StateType >
Iir::PoleFilter< BaseClass, StateType, MaxAnalogPoles, MaxDigitalPoles >

Public Member Functions

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.35.1 Detailed Description

template<int MaxStages, class StateType>
class Iir::CascadeStages< MaxStages, StateType >

Storage for Cascade: This holds a chain of 2nd order filters with its coefficients.

7.35.2 Member Function Documentation

filter()

```
template<int MaxStages, class StateType >
template<typename Sample >
Sample Iir::CascadeStages< MaxStages, StateType >::filter (
            const Sample in )  [inline]
```
Filters one sample through the whole chain of biquads and return the result

Parameters

in	Sample to be filtered

Returns

filtered sample

getCascadeStorage()

```
template<int MaxStages, class StateType >
const Cascade::Storage Iir::CascadeStages< MaxStages, StateType >::getCascadeStorage ( )
[inline]
```
Returns the coefficients of the entire Biquad chain

reset()

```
template<int MaxStages, class StateType >
```

```
void Iir::CascadeStages< MaxStages, StateType >::reset ( )  [inline]
```
Resets all biquads (i.e. the delay lines but not the coefficients)

setup()

```
template<int MaxStages, class StateType >
void Iir::CascadeStages< MaxStages, StateType >::setup (
             const double(&) sosCoefficients[MaxStages][6] )  [inline]
```
Sets the coefficients of the whole chain of biquads.

Parameters

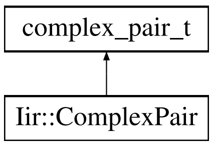
sosCoefficients	2D array in Python style sos ordering: 0-2: FIR, 3-5: IIR coeff.

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

- iir/Cascade.h

7.36 Iir::ComplexPair Struct Reference

```
#include <Types.h>
```
Inheritance diagram for Iir::ComplexPair:



Public Member Functions

- bool isMatchedPair () const

7.36.1 Detailed Description

A conjugate or real pair

7.36.2 Member Function Documentation

isMatchedPair()

```
bool Iir::ComplexPair::isMatchedPair ( ) const  [inline]
```
Returns true if this is either a conjugate pair, or a pair of reals where neither is zero.
The documentation for this struct was generated from the following file:

- iir/Types.h

7.37 Iir::DirectFormI Class Reference

```
#include <State.h>
```

7.37.1 Detailed Description

State for applying a second order section to a sample using Direct Form I
Difference equation:
y[n] = (b0/a0)∗x[n] + (b1/a0)∗x[n-1] + (b2/a0)∗x[n-2]

- (a1/a0)∗y[n-1] - (a2/a0)∗y[n-2]

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

- iir/State.h

7.38 Iir::DirectFormII Class Reference

```
#include <State.h>
```

7.38.1 Detailed Description

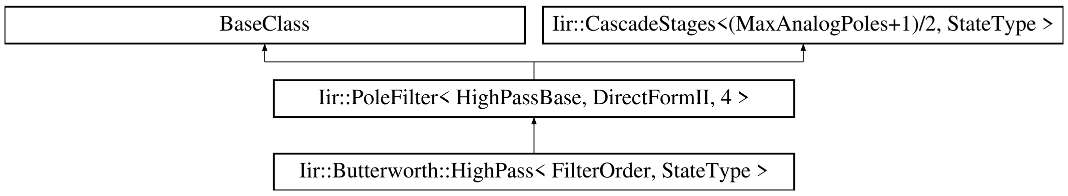
State for applying a second order section to a sample using Direct Form II
Difference equation:
v[n] = x[n] - (a1/a0)∗v[n-1] - (a2/a0)∗v[n-2] y(n) = (b0/a0)∗v[n] + (b1/a0)∗v[n-1] + (b2/a0)∗v[n-2]
The documentation for this class was generated from the following file:

- iir/State.h

7.39 Iir::Butterworth::HighPass< FilterOrder, StateType > Struct Template Reference

```
#include <Butterworth.h>
```

Inheritance diagram for Iir::Butterworth::HighPass< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< HighPassBase, DirectFormII, 4 >
Iir::Butterworth::HighPass< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double cutoffFrequency)
- void setup (int reqOrder, double sampleRate, double cutoffFrequency)
- void setupN (double cutoffFrequency)
- void setupN (int reqOrder, double cutoffFrequency)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.39.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::Butterworth::HighPass< FilterOrder, StateType >

Butterworth Highpass filter.

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.39.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::HighPass< FilterOrder, StateType >::setup (
             double sampleRate,
             double cutoffFrequency )  [inline]
```

Calculates the coefficients with the filter order provided by the instantiation

Parameters

sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::HighPass< FilterOrder, StateType >::setup (
             int reqOrder,
             double sampleRate,
             double cutoffFrequency )  [inline]
```

Calculates the coefficients

Parameters

reqOrder	The actual order which can be less than the instantiated one
sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::HighPass< FilterOrder, StateType >::setupN (
             double cutoffFrequency )  [inline]
```

Calculates the coefficients with the filter order provided by the instantiation

Parameters

cutoffFrequency	Normalised cutoff frequency (0..1/2)

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::HighPass< FilterOrder, StateType >::setupN (
             int reqOrder,
             double cutoffFrequency )  [inline]
```

Calculates the coefficients

Parameters

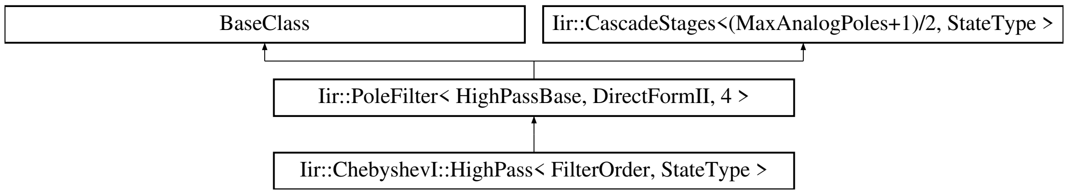
reqOrder	The actual order which can be less than the instantiated one
cutoffFrequency	Normalised cutoff frequency (0..1/2)

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

- iir/Butterworth.h

7.40 Iir::ChebyshevI::HighPass< FilterOrder, StateType > Struct Template Reference

`#include <ChebyshevI.h>`
Inheritance diagram for Iir::ChebyshevI::HighPass< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< HighPassBase, DirectFormII, 4 >
Iir::ChebyshevI::HighPass< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double cutoffFrequency, double rippleDb)
- void setup (int reqOrder, double sampleRate, double cutoffFrequency, double rippleDb)
- void setupN (double cutoffFrequency, double rippleDb)
- void setupN (int reqOrder, double cutoffFrequency, double rippleDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.40.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::ChebyshevI::HighPass< FilterOrder, StateType >

ChebyshevI highpass filter

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.40.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::HighPass< FilterOrder, StateType >::setup (
              double sampleRate,
              double cutoffFrequency,
              double rippleDb ) [inline]
```

Calculates the coefficients of the filter at the order FilterOrder

Parameters

sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency.
rippleDb	Permitted ripples in dB in the passband

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::HighPass< FilterOrder, StateType >::setup (
             int reqOrder,
             double sampleRate,
             double cutoffFrequency,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at specified order

Parameters

reqOrder	Actual order for the filter calculations
sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency.
rippleDb	Permitted ripples in dB in the passband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::HighPass< FilterOrder, StateType >::setupN (
             double cutoffFrequency,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at the order FilterOrder

Parameters

cutoffFrequency	Normalised cutoff frequency (0..1/2)
rippleDb	Permitted ripples in dB in the passband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::HighPass< FilterOrder, StateType >::setupN (
             int reqOrder,
             double cutoffFrequency,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at specified order

Parameters

reqOrder	Actual order for the filter calculations
cutoffFrequency	Normalised cutoff frequency (0..1/2)
rippleDb	Permitted ripples in dB in the passband

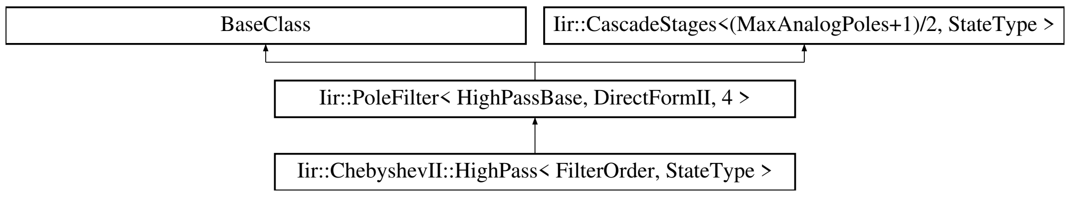
The documentation for this struct was generated from the following file:

- iir/ChebyshevI.h

7.41 Iir::ChebyshevII::HighPass< FilterOrder, StateType > Struct Template Reference

```
#include <ChebyshevII.h>
```

Inheritance diagram for Iir::ChebyshevII::HighPass< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< HighPassBase, DirectFormII, 4 >
Iir::ChebyshevII::HighPass< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double cutoffFrequency, double stopBandDb)
- void setup (int reqOrder, double sampleRate, double cutoffFrequency, double stopBandDb)
- void setupN (double cutoffFrequency, double stopBandDb)
- void setupN (int reqOrder, double cutoffFrequency, double stopBandDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.41.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::ChebyshevII::HighPass< FilterOrder, StateType >

ChebyshevII highpass filter

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.41.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::HighPass< FilterOrder, StateType >::setup (
             double sampleRate,
             double cutoffFrequency,
             double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency.
stopBandDb	Permitted ripples in dB in the stopband

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::HighPass< FilterOrder, StateType >::setup (
             int reqOrder,
             double sampleRate,
             double cutoffFrequency,
             double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

reqOrder	Requested order which can be less than the instantiated one
sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency.
stopBandDb	Permitted ripples in dB in the stopband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::HighPass< FilterOrder, StateType >::setupN (
            double cutoffFrequency,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

cutoffFrequency	Normalised cutoff frequency (0..1/2)
stopBandDb	Permitted ripples in dB in the stopband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::HighPass< FilterOrder, StateType >::setupN (
            int reqOrder,
            double cutoffFrequency,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

reqOrder	Requested order which can be less than the instantiated one
cutoffFrequency	Normalised cutoff frequency (0..1/2)
stopBandDb	Permitted ripples in dB in the stopband

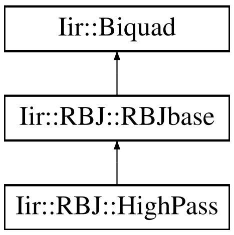
The documentation for this struct was generated from the following file:

- iir/ChebyshevII.h

7.42 Iir::RBJ::HighPass Struct Reference

```
#include <RBJ.h>
```

Inheritance diagram for Iir::RBJ::HighPass:



Public Member Functions

- void setupN (double cutoffFrequency, double q=(1/sqrt(2)))
- void setup (double sampleRate, double cutoffFrequency, double q=(1/sqrt(2)))

Public Member Functions inherited from Iir::RBJ::RBJbase

- template<typename Sample >
  Sample **filter** (Sample s)

  filter operation
- void **reset** ()

  resets the delay lines to zero
- const DirectFormI & **getState** ()

  gets the delay lines (=state) of the filter

Public Member Functions inherited from Iir::Biquad

- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const
- double getA0 () const
- double getA1 () const
- double getA2 () const
- double getB0 () const
- double getB1 () const
- double getB2 () const
- template<class StateType >
  double filter (double s, StateType &state) const
- void setCoefficients (double a0, double a1, double a2, double b0, double b1, double b2)
- void setOnePole (complex_t pole, complex_t zero)
- void setTwoPole (complex_t pole1, complex_t zero1, complex_t pole2, complex_t zero2)
- void setPoleZeroPair (const PoleZeroPair &pair)
- void setIdentity ()
- void applyScale (double scale)

7.42.1 Detailed Description

Highpass.

7.42.2 Member Function Documentation

setup()

```
void Iir::RBJ::HighPass::setup (
             double sampleRate,
             double cutoffFrequency,
             double q = (1/sqrt(2)) )   [inline]
```

Calculates the coefficients

Parameters

sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency
q	Q factor determines the resonance peak at the cutoff.

setupN()

```
void Iir::RBJ::HighPass::setupN (
             double cutoffFrequency,
             double q = (1/sqrt(2)) )
```

Calculates the coefficients

Parameters

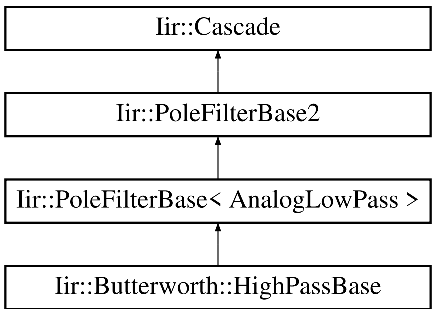
cutoffFrequency	Normalised cutoff frequency (0..1/2)
q	Q factor determines the resonance peak at the cutoff.

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

- iir/RBJ.h
- iir/RBJ.cpp

7.43 Iir::Butterworth::HighPassBase Struct Reference

Inheritance diagram for Iir::Butterworth::HighPassBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowPass >
Iir::Butterworth::HighPassBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

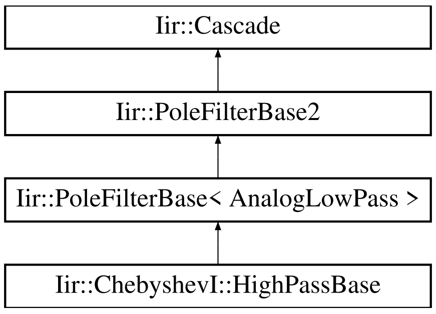
- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/Butterworth.h
- iir/Butterworth.cpp

7.44 Iir::ChebyshevI::HighPassBase Struct Reference

Inheritance diagram for Iir::ChebyshevI::HighPassBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowPass >
Iir::ChebyshevI::HighPassBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

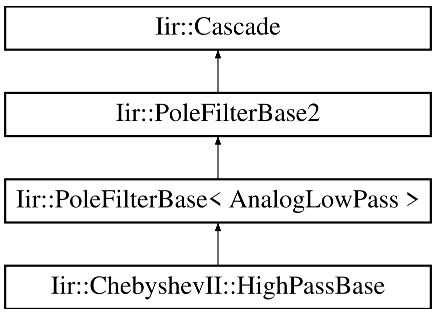
- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/ChebyshevI.h
- iir/ChebyshevI.cpp

7.45 Iir::ChebyshevII::HighPassBase Struct Reference

Inheritance diagram for Iir::ChebyshevII::HighPassBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowPass >
Iir::ChebyshevII::HighPassBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/ChebyshevII.h
- iir/ChebyshevII.cpp

7.46 Iir::HighPassTransform Class Reference

```
#include <PoleFilter.h>
```

7.46.1 Detailed Description

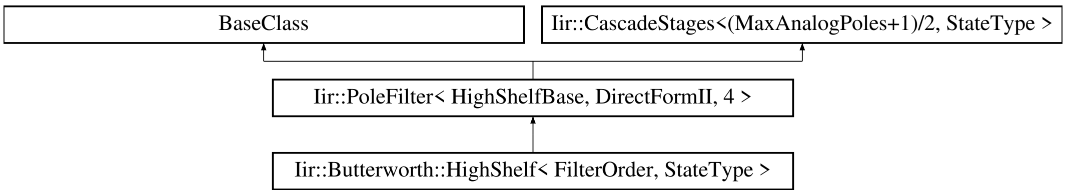
low pass to high pass
The documentation for this class was generated from the following files:

- iir/PoleFilter.h
- iir/PoleFilter.cpp

7.47 Iir::Butterworth::HighShelf< FilterOrder, StateType > Struct Template Reference

```
#include <Butterworth.h>
```

Inheritance diagram for Iir::Butterworth::HighShelf< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< HighShelfBase, DirectFormII, 4 >
Iir::Butterworth::HighShelf< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double cutoffFrequency, double gainDb)
- void setup (int reqOrder, double sampleRate, double cutoffFrequency, double gainDb)
- void setupN (double cutoffFrequency, double gainDb)
- void setupN (int reqOrder, double cutoffFrequency, double gainDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.47.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::Butterworth::HighShelf< FilterOrder, StateType >

Butterworth high shelf filter. Above the cutoff the filter has a specified gain and below it has 0 dB.

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.47.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::HighShelf< FilterOrder, StateType >::setup (
             double sampleRate,
             double cutoffFrequency,
             double gainDb ) [inline]
```

Calculates the coefficients with the filter order provided by the instantiation

Parameters

sampleRate	Sampling rate
cutoffFrequency	Cutoff
gainDb	Gain in dB of the filter in the passband

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::HighShelf< FilterOrder, StateType >::setup (
             int reqOrder,
             double sampleRate,
             double cutoffFrequency,
             double gainDb )  [inline]
```

Calculates the coefficients

Parameters

reqOrder	The actual order which can be less than the instantiated one
sampleRate	Sampling rate
cutoffFrequency	Cutoff
gainDb	Gain in dB of the filter in the passband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::HighShelf< FilterOrder, StateType >::setupN (
             double cutoffFrequency,
             double gainDb )  [inline]
```

Calculates the coefficients with the filter order provided by the instantiation

Parameters

cutoffFrequency	Normalised cutoff frequency (0..1/2)
gainDb	Gain in dB of the filter in the passband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::HighShelf< FilterOrder, StateType >::setupN (
             int reqOrder,
             double cutoffFrequency,
             double gainDb )  [inline]
```

Calculates the coefficients

Parameters

reqOrder	The actual order which can be less than the instantiated one
cutoffFrequency	Normalised cutoff frequency (0..1/2)
gainDb	Gain in dB of the filter in the passband

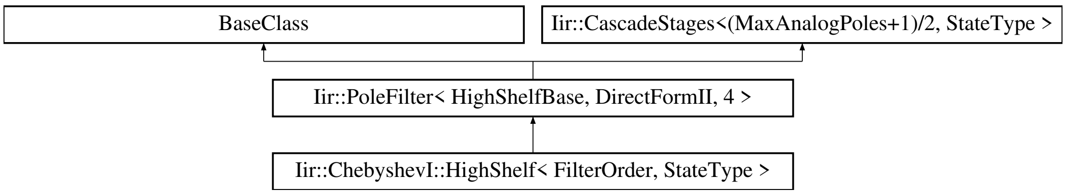
The documentation for this struct was generated from the following file:

- iir/Butterworth.h

7.48 Iir::ChebyshevI::HighShelf< FilterOrder, StateType > Struct Template Reference

`#include <ChebyshevI.h>`

Inheritance diagram for Iir::ChebyshevI::HighShelf< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< HighShelfBase, DirectFormII, 4 >
Iir::ChebyshevI::HighShelf< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double cutoffFrequency, double gainDb, double rippleDb)
- void setup (int reqOrder, double sampleRate, double cutoffFrequency, double gainDb, double rippleDb)
- void setupN (double cutoffFrequency, double gainDb, double rippleDb)
- void setupN (int reqOrder, double cutoffFrequency, double gainDb, double rippleDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.48.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::ChebyshevI::HighShelf< FilterOrder, StateType >

ChebyshevI high shelf filter. Specified gain in the passband. Otherwise 0 dB.

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.48.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::HighShelf< FilterOrder, StateType >::setup (
             double sampleRate,
             double cutoffFrequency,
             double gainDb,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at the order FilterOrder

Parameters

sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency.
gainDb	Gain in the passband
rippleDb	Permitted ripples in dB in the passband

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::HighShelf< FilterOrder, StateType >::setup (
            int reqOrder,
            double sampleRate,
            double cutoffFrequency,
            double gainDb,
            double rippleDb )  [inline]
```

Calculates the coefficients of the filter at specified order

Parameters

reqOrder	Actual order for the filter calculations
sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency.
gainDb	Gain in the passband
rippleDb	Permitted ripples in dB in the passband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::HighShelf< FilterOrder, StateType >::setupN (
            double cutoffFrequency,
            double gainDb,
            double rippleDb )  [inline]
```

Calculates the coefficients of the filter at the order FilterOrder

Parameters

cutoffFrequency	Normalised cutoff frequency (0..1/2)
gainDb	Gain in the passband
rippleDb	Permitted ripples in dB in the passband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::HighShelf< FilterOrder, StateType >::setupN (
            int reqOrder,
            double cutoffFrequency,
            double gainDb,
            double rippleDb )  [inline]
```

Calculates the coefficients of the filter at specified order

Parameters

reqOrder	Actual order for the filter calculations
cutoffFrequency	Normalised cutoff frequency (0..1/2)
gainDb	Gain in the passband
rippleDb	Permitted ripples in dB in the passband

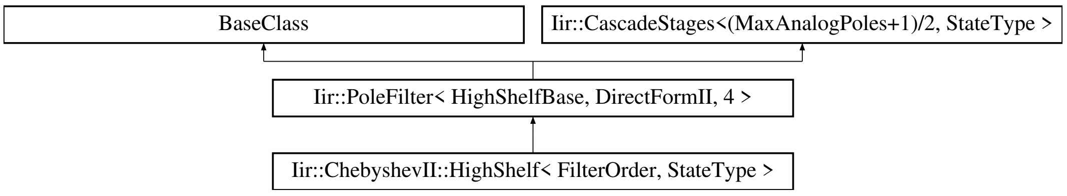
The documentation for this struct was generated from the following file:

- iir/ChebyshevI.h

7.49 Iir::ChebyshevII::HighShelf< FilterOrder, StateType > Struct Template Reference

`#include <ChebyshevII.h>`

Inheritance diagram for Iir::ChebyshevII::HighShelf< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< HighShelfBase, DirectFormII, 4 >
Iir::ChebyshevII::HighShelf< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double cutoffFrequency, double gainDb, double stopBandDb)
- void setup (int reqOrder, double sampleRate, double cutoffFrequency, double gainDb, double stopBandDb)
- void setupN (double cutoffFrequency, double gainDb, double stopBandDb)
- void setupN (int reqOrder, double cutoffFrequency, double gainDb, double stopBandDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.49.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::ChebyshevII::HighShelf< FilterOrder, StateType >

ChebyshevII high shelf filter. Specified gain in the passband and 0dB in the stopband.

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.49.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::HighShelf< FilterOrder, StateType >::setup (
            double sampleRate,
            double cutoffFrequency,
            double gainDb,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency.
gainDb	Gain the passbard. The stopband has 0 dB gain.
stopBandDb	Permitted ripples in dB in the stopband

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::HighShelf< FilterOrder, StateType >::setup (
             int reqOrder,
             double sampleRate,
             double cutoffFrequency,
             double gainDb,
             double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

reqOrder	Requested order which can be less than the instantiated one
sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency.
gainDb	Gain the passbard. The stopband has 0 dB gain.
stopBandDb	Permitted ripples in dB in the stopband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::HighShelf< FilterOrder, StateType >::setupN (
             double cutoffFrequency,
             double gainDb,
             double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

cutoffFrequency	Normalised cutoff frequency (0..1/2)
gainDb	Gain the passbard. The stopband has 0 dB gain.
stopBandDb	Permitted ripples in dB in the stopband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::HighShelf< FilterOrder, StateType >::setupN (
             int reqOrder,
             double cutoffFrequency,
             double gainDb,
             double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

reqOrder	Requested order which can be less than the instantiated one
cutoffFrequency	Normalised cutoff frequency (0..1/2)
gainDb	Gain the passbard. The stopband has 0 dB gain.
stopBandDb	Permitted ripples in dB in the stopband

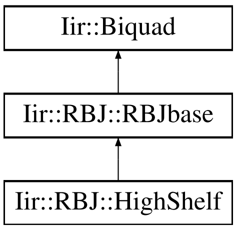
The documentation for this struct was generated from the following file:

- iir/ChebyshevII.h

7.50 Iir::RBJ::HighShelf Struct Reference

```
#include <RBJ.h>
```

Inheritance diagram for Iir::RBJ::HighShelf:



Public Member Functions

- void setupN (double cutoffFrequency, double gainDb, double shelfSlope=1)
- void setup (double sampleRate, double cutoffFrequency, double gainDb, double shelfSlope=1)

Public Member Functions inherited from Iir::RBJ::RBJbase

- template<typename Sample >
  Sample **filter** (Sample s)

  filter operation
- void **reset** ()

  resets the delay lines to zero
- const DirectFormI & **getState** ()

  gets the delay lines (=state) of the filter

Public Member Functions inherited from Iir::Biquad

- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const
- double getA0 () const
- double getA1 () const
- double getA2 () const
- double getB0 () const
- double getB1 () const
- double getB2 () const
- template<class StateType >
  double filter (double s, StateType &state) const
- void setCoefficients (double a0, double a1, double a2, double b0, double b1, double b2)
- void setOnePole (complex_t pole, complex_t zero)
- void setTwoPole (complex_t pole1, complex_t zero1, complex_t pole2, complex_t zero2)
- void setPoleZeroPair (const PoleZeroPair &pair)
- void setIdentity ()
- void applyScale (double scale)

7.50.1 Detailed Description

High shelf: 0db in the stopband and gainDb in the passband.

7.50.2 Member Function Documentation

setup()

```
void Iir::RBJ::HighShelf::setup (
            double sampleRate,
            double cutoffFrequency,
            double gainDb,
            double shelfSlope = 1 )  [inline]
```

Calculates the coefficients

Parameters

sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency
gainDb	Gain in the passband
shelfSlope	Slope between stop/passband. 1 = as steep as it can.

setupN()

```
void Iir::RBJ::HighShelf::setupN (
            double cutoffFrequency,
            double gainDb,
            double shelfSlope = 1 )
```

Calculates the coefficients

Parameters

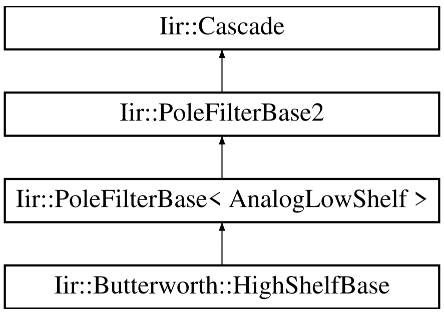
cutoffFrequency	Normalised cutoff frequency
gainDb	Gain in the passband
shelfSlope	Slope between stop/passband. 1 = as steep as it can.

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

- iir/RBJ.h
- iir/RBJ.cpp

7.51 Iir::Butterworth::HighShelfBase Struct Reference

Inheritance diagram for Iir::Butterworth::HighShelfBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowShelf >
Iir::Butterworth::HighShelfBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

- int getNumStages () const

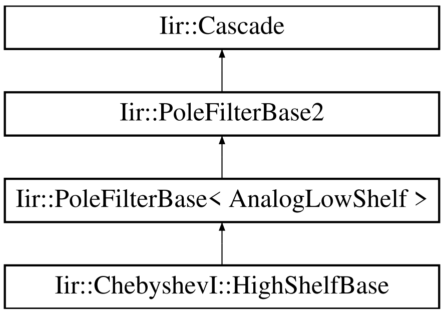
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/Butterworth.h
- iir/Butterworth.cpp

7.52 Iir::ChebyshevI::HighShelfBase Struct Reference

Inheritance diagram for Iir::ChebyshevI::HighShelfBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowShelf >
Iir::ChebyshevI::HighShelfBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

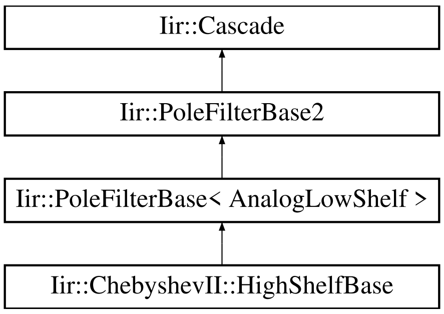
- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/ChebyshevI.h
- iir/ChebyshevI.cpp

7.53 Iir::ChebyshevII::HighShelfBase Struct Reference

Inheritance diagram for Iir::ChebyshevII::HighShelfBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowShelf >
Iir::ChebyshevII::HighShelfBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

- int getNumStages () const
- const Biquad & operator[ ] (int index)

- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

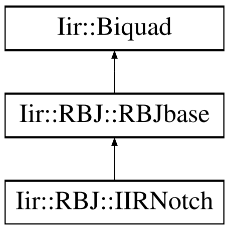
The documentation for this struct was generated from the following files:

- iir/ChebyshevII.h
- iir/ChebyshevII.cpp

7.54 Iir::RBJ::IIRNotch Struct Reference

`#include <RBJ.h>`

Inheritance diagram for Iir::RBJ::IIRNotch:



Public Member Functions

- void setupN (double centerFrequency, double q_factor=10)
- void setup (double sampleRate, double centerFrequency, double q_factor=10)

Public Member Functions inherited from Iir::RBJ::RBJbase

- template<typename Sample >
  Sample **filter** (Sample s)

  filter operation
- void **reset** ()

  resets the delay lines to zero
- const DirectFormI & **getState** ()

  gets the delay lines (=state) of the filter

Public Member Functions inherited from Iir::Biquad

- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const
- double getA0 () const
- double getA1 () const
- double getA2 () const
- double getB0 () const
- double getB1 () const
- double getB2 () const
- template<class StateType >
  double filter (double s, StateType &state) const
- void setCoefficients (double a0, double a1, double a2, double b0, double b1, double b2)
- void setOnePole (complex_t pole, complex_t zero)
- void setTwoPole (complex_t pole1, complex_t zero1, complex_t pole2, complex_t zero2)
- void setPoleZeroPair (const PoleZeroPair &pair)
- void setIdentity ()
- void applyScale (double scale)

7.54.1 Detailed Description

Bandstop with Q factor: the higher the Q factor the more narrow is the notch. However, a narrow notch has a long impulse response (= ringing) and numerical problems might prevent perfect damping. Practical values of the Q factor are about Q = 10 to 20. In terms of the design the Q factor defines the radius of the poles as r = exp(-pi∗(centerFrequency/sampleRate)/q_factor) whereas the angles of the poles/zeros define the bandstop frequency. The higher Q the closer r moves towards the unit circle.

7.54.2 Member Function Documentation

setup()

```
void Iir::RBJ::IIRNotch::setup (
            double sampleRate,
            double centerFrequency,
            double q_factor = 10 )  [inline]
```

Calculates the coefficients

Parameters

sampleRate	Sampling rate
centerFrequency	Center frequency of the notch
q_factor	Q factor of the notch (1 to ∼20)

setupN()

```
void Iir::RBJ::IIRNotch::setupN (
            double centerFrequency,
            double q_factor = 10 )
```

Calculates the coefficients

Parameters

centerFrequency	Normalised centre frequency of the notch
q_factor	Q factor of the notch (1 to ∼20)

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

- iir/RBJ.h
- iir/RBJ.cpp

7.55 Iir::Layout< MaxPoles > Class Template Reference

```
#include <Layout.h>
```

7.55.1 Detailed Description

template<int MaxPoles>
class Iir::Layout< MaxPoles >

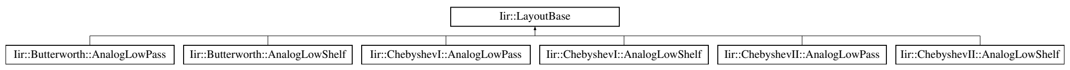
Storage for Layout
The documentation for this class was generated from the following file:

- iir/Layout.h

7.56 Iir::LayoutBase Class Reference

```
#include <Layout.h>
```

Inheritance diagram for Iir::LayoutBase:


Iir::LayoutBase
Iir::Butterworth::AnalogLowPass
Iir::Butterworth::AnalogLowShelf
Iir::ChebyshevI::AnalogLowPass
Iir::ChebyshevI::AnalogLowShelf
Iir::ChebyshevII::AnalogLowPass
Iir::ChebyshevII::AnalogLowShelf

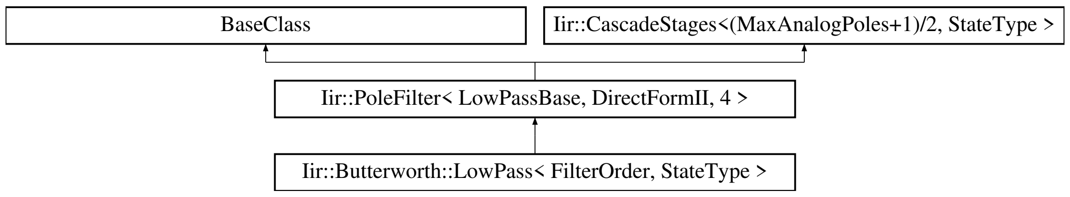
7.56.1 Detailed Description

Base uses pointers to reduce template instantiations
The documentation for this class was generated from the following file:

- iir/Layout.h

7.57 Iir::Butterworth::LowPass< FilterOrder, StateType > Struct Template Reference

`#include <Butterworth.h>`
Inheritance diagram for Iir::Butterworth::LowPass< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< LowPassBase, DirectFormII, 4 >
Iir::Butterworth::LowPass< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double cutoffFrequency)
- void setup (int reqOrder, double sampleRate, double cutoffFrequency)
- void setupN (double cutoffFrequency)
- void setupN (int reqOrder, double cutoffFrequency)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.57.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::Butterworth::LowPass< FilterOrder, StateType >

Butterworth Lowpass filter.

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.57.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::LowPass< FilterOrder, StateType >::setup (
          double sampleRate,
            double cutoffFrequency )  [inline]
```

Calculates the coefficients

Parameters

sampleRate	Sampling rate
cutoffFrequency	Cutoff

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::LowPass< FilterOrder, StateType >::setup (
            int reqOrder,
            double sampleRate,
            double cutoffFrequency )  [inline]
```

Calculates the coefficients

Parameters

reqOrder	The actual order which can be less than the instantiated one
sampleRate	Sampling rate
cutoffFrequency	Cutoff

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::LowPass< FilterOrder, StateType >::setupN (
            double cutoffFrequency )  [inline]
```

Calculates the coefficients

Parameters

cutoffFrequency	Normalised cutoff frequency (0..1/2)

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::LowPass< FilterOrder, StateType >::setupN (
            int reqOrder,
            double cutoffFrequency )  [inline]
```

Calculates the coefficients

Parameters

reqOrder	The actual order which can be less than the instantiated one
cutoffFrequency	Normalised cutoff frequency (0..1/2)

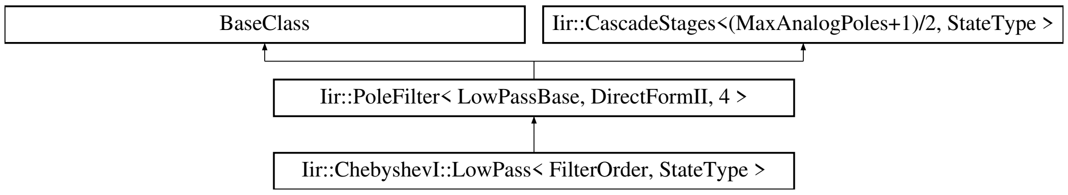
The documentation for this struct was generated from the following file:

- iir/Butterworth.h

7.58 Iir::ChebyshevI::LowPass< FilterOrder, StateType > Struct Template Reference

```
#include <ChebyshevI.h>
```

Inheritance diagram for Iir::ChebyshevI::LowPass< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< LowPassBase, DirectFormII, 4 >
Iir::ChebyshevI::LowPass< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double cutoffFrequency, double rippleDb)
- void setup (int reqOrder, double sampleRate, double cutoffFrequency, double rippleDb)
- void setupN (double cutoffFrequency, double rippleDb)
- void setupN (int reqOrder, double cutoffFrequency, double rippleDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.58.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::ChebyshevI::LowPass< FilterOrder, StateType >

ChebyshevI lowpass filter

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.58.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::LowPass< FilterOrder, StateType >::setup (
             double sampleRate,
             double cutoffFrequency,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at the order FilterOrder

Parameters

sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency
rippleDb	Permitted ripples in dB in the passband

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::LowPass< FilterOrder, StateType >::setup (
             int reqOrder,
             double sampleRate,
             double cutoffFrequency,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at specified order

Parameters

reqOrder	Actual order for the filter calculations
sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency.
rippleDb	Permitted ripples in dB in the passband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::LowPass< FilterOrder, StateType >::setupN (
            double cutoffFrequency,
            double rippleDb )  [inline]
```

Calculates the coefficients of the filter at the order FilterOrder

Parameters

cutoffFrequency	Normalised cutoff frequency (0..1/2)
rippleDb	Permitted ripples in dB in the passband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::LowPass< FilterOrder, StateType >::setupN (
            int reqOrder,
            double cutoffFrequency,
            double rippleDb )  [inline]
```

Calculates the coefficients of the filter at specified order

Parameters

reqOrder	Actual order for the filter calculations
cutoffFrequency	Normalised cutoff frequency (0..1/2)
rippleDb	Permitted ripples in dB in the passband

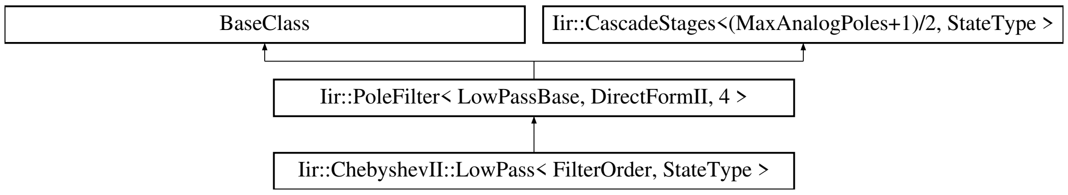
The documentation for this struct was generated from the following file:

- iir/ChebyshevI.h

7.59 Iir::ChebyshevII::LowPass< FilterOrder, StateType > Struct Template Reference

```
#include <ChebyshevII.h>
```

Inheritance diagram for Iir::ChebyshevII::LowPass< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< LowPassBase, DirectFormII, 4 >
Iir::ChebyshevII::LowPass< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double cutoffFrequency, double stopBandDb)
- void setup (int reqOrder, double sampleRate, double cutoffFrequency, double stopBandDb)
- void setupN (double cutoffFrequency, double stopBandDb)
- void setupN (int reqOrder, double cutoffFrequency, double stopBandDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.59.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::ChebyshevII::LowPass< FilterOrder, StateType >

ChebyshevII lowpass filter

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.59.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::LowPass< FilterOrder, StateType >::setup (
            double sampleRate,
            double cutoffFrequency,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency.
stopBandDb	Permitted ripples in dB in the stopband

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::LowPass< FilterOrder, StateType >::setup (
             int reqOrder,
             double sampleRate,
             double cutoffFrequency,
             double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

reqOrder	Requested order which can be less than the instantiated one
sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency.
stopBandDb	Permitted ripples in dB in the stopband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::LowPass< FilterOrder, StateType >::setupN (
            double cutoffFrequency,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

cutoffFrequency	Normalised cutoff frequency (0..1/2)
stopBandDb	Permitted ripples in dB in the stopband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::LowPass< FilterOrder, StateType >::setupN (
            int reqOrder,
            double cutoffFrequency,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

reqOrder	Requested order which can be less than the instantiated one
cutoffFrequency	Normalised cutoff frequency (0..1/2)
stopBandDb	Permitted ripples in dB in the stopband

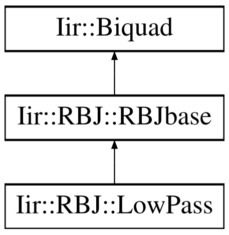
The documentation for this struct was generated from the following file:

- iir/ChebyshevII.h

7.60 Iir::RBJ::LowPass Struct Reference

`#include <RBJ.h>`

Inheritance diagram for Iir::RBJ::LowPass:



Public Member Functions

- void setupN (double cutoffFrequency, double q=(1/sqrt(2)))
- void setup (double sampleRate, double cutoffFrequency, double q=(1/sqrt(2)))

Public Member Functions inherited from Iir::RBJ::RBJbase

- template<typename Sample >
  Sample **filter** (Sample s)

  filter operation
- void **reset** ()

  resets the delay lines to zero
- const DirectFormI & **getState** ()

  gets the delay lines (=state) of the filter

Public Member Functions inherited from Iir::Biquad

- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const
- double getA0 () const
- double getA1 () const
- double getA2 () const
- double getB0 () const
- double getB1 () const
- double getB2 () const
- template<class StateType >
  double filter (double s, StateType &state) const
- void setCoefficients (double a0, double a1, double a2, double b0, double b1, double b2)
- void setOnePole (complex_t pole, complex_t zero)
- void setTwoPole (complex_t pole1, complex_t zero1, complex_t pole2, complex_t zero2)
- void setPoleZeroPair (const PoleZeroPair &pair)
- void setIdentity ()
- void applyScale (double scale)

7.60.1 Detailed Description

Lowpass.

7.60.2 Member Function Documentation

setup()

```
void Iir::RBJ::LowPass::setup (
            double sampleRate,
            double cutoffFrequency,
            double q = (1/sqrt(2)) )  [inline]
```

Calculates the coefficients

Parameters

sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency
q	Q factor determines the resonance peak at the cutoff.

setupN()

```
void Iir::RBJ::LowPass::setupN (
            double cutoffFrequency,
            double q = (1/sqrt(2)) )
```

Calculates the coefficients

Parameters

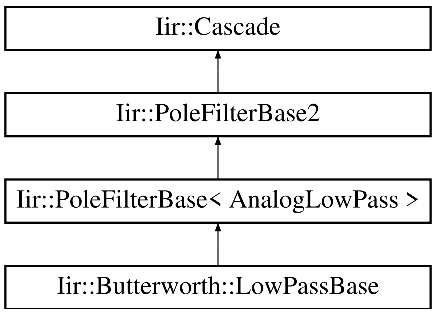
cutoffFrequency	Normalised cutoff frequency
q	Q factor determines the resonance peak at the cutoff.

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

- iir/RBJ.h
- iir/RBJ.cpp

7.61 Iir::Butterworth::LowPassBase Struct Reference

Inheritance diagram for Iir::Butterworth::LowPassBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowPass >
Iir::Butterworth::LowPassBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

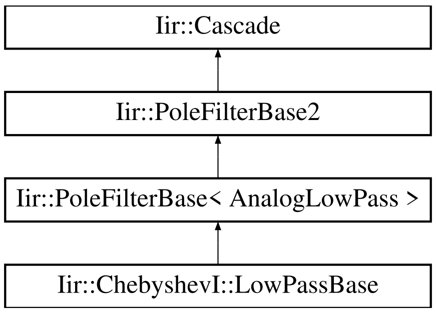
- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/Butterworth.h
- iir/Butterworth.cpp

7.62 Iir::ChebyshevI::LowPassBase Struct Reference

Inheritance diagram for Iir::ChebyshevI::LowPassBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowPass >
Iir::ChebyshevI::LowPassBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

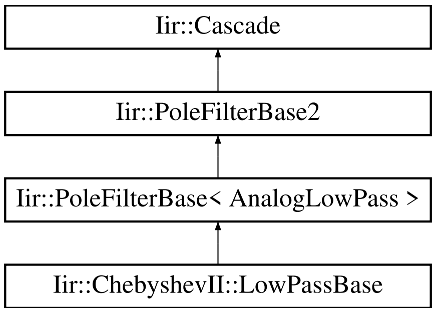
- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/ChebyshevI.h
- iir/ChebyshevI.cpp

7.63 Iir::ChebyshevII::LowPassBase Struct Reference

Inheritance diagram for Iir::ChebyshevII::LowPassBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowPass >
Iir::ChebyshevII::LowPassBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/ChebyshevII.h
- iir/ChebyshevII.cpp

7.64 Iir::LowPassTransform Class Reference

```
#include <PoleFilter.h>
```

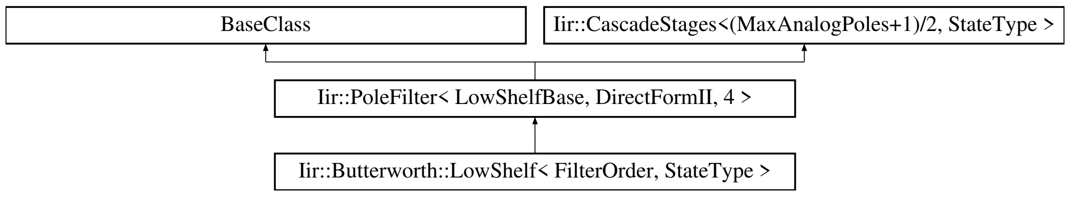
7.64.1 Detailed Description

s-plane to z-plane transforms
For pole filters, an analog prototype is created via placement of poles and zeros in the s-plane. The analog prototype is either a halfband low pass or a halfband low shelf. The poles, zeros, and normalization parameters are transformed into the z-plane using variants of the bilinear transformation. low pass to low pass
The documentation for this class was generated from the following files:

- iir/PoleFilter.h
- iir/PoleFilter.cpp

7.65 Iir::Butterworth::LowShelf< FilterOrder, StateType > Struct Template Reference

`#include <Butterworth.h>`
Inheritance diagram for Iir::Butterworth::LowShelf< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< LowShelfBase, DirectFormII, 4 >
Iir::Butterworth::LowShelf< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double cutoffFrequency, double gainDb)
- void setup (int reqOrder, double sampleRate, double cutoffFrequency, double gainDb)
- void setupN (double cutoffFrequency, double gainDb)
- void setupN (int reqOrder, double cutoffFrequency, double gainDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.65.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::Butterworth::LowShelf< FilterOrder, StateType >

Butterworth low shelf filter: below the cutoff it has a specified gain and above the cutoff the gain is 0 dB.

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.65.2 Member Function Documentation

setup() `[1/2]`

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::LowShelf< FilterOrder, StateType >::setup (
            double sampleRate,
            double cutoffFrequency,
            double gainDb )  [inline]
```

Calculates the coefficients with the filter order provided by the instantiation

Parameters

sampleRate	Sampling rate
cutoffFrequency	Cutoff
gainDb	Gain in dB of the filter in the passband

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::LowShelf< FilterOrder, StateType >::setup (
            int reqOrder,
            double sampleRate,
            double cutoffFrequency,
            double gainDb )  [inline]
```

Calculates the coefficients

Parameters

reqOrder	The actual order which can be less than the instantiated one
sampleRate	Sampling rate
cutoffFrequency	Cutoff
gainDb	Gain in dB of the filter in the passband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::LowShelf< FilterOrder, StateType >::setupN (
            double cutoffFrequency,
            double gainDb )  [inline]
```

Calculates the coefficients with the filter order provided by the instantiation

Parameters

cutoffFrequency	Normalised cutoff frequency (0..1/2)
gainDb	Gain in dB of the filter in the passband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::Butterworth::LowShelf< FilterOrder, StateType >::setupN (
            int reqOrder,
            double cutoffFrequency,
            double gainDb )  [inline]
```

Calculates the coefficients

Parameters

reqOrder	The actual order which can be less than the instantiated one
cutoffFrequency	Normalised cutoff frequency (0..1/2)
gainDb	Gain in dB of the filter in the passband

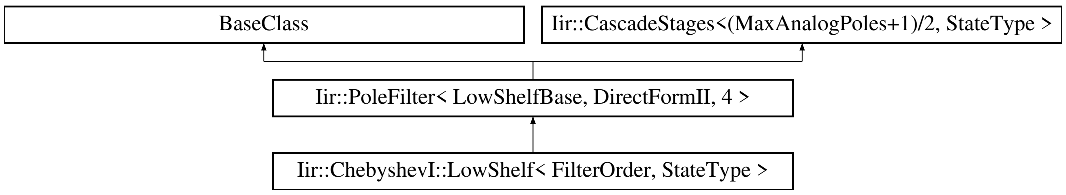
The documentation for this struct was generated from the following file:

- iir/Butterworth.h

7.66 Iir::ChebyshevI::LowShelf< FilterOrder, StateType > Struct Template Reference

`#include <ChebyshevI.h>`

Inheritance diagram for Iir::ChebyshevI::LowShelf< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< LowShelfBase, DirectFormII, 4 >
Iir::ChebyshevI::LowShelf< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double cutoffFrequency, double gainDb, double rippleDb)
- void setup (int reqOrder, double sampleRate, double cutoffFrequency, double gainDb, double rippleDb)
- void setupN (double cutoffFrequency, double gainDb, double rippleDb)
- void setupN (int reqOrder, double cutoffFrequency, double gainDb, double rippleDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.66.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::ChebyshevI::LowShelf< FilterOrder, StateType >

ChebyshevI low shelf filter. Specified gain in the passband. Otherwise 0 dB.

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.66.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::LowShelf< FilterOrder, StateType >::setup (
             double sampleRate,
             double cutoffFrequency,
             double gainDb,
             double rippleDb )  [inline]
```

Calculates the coefficients of the filter at the order FilterOrder

Parameters

sampleRate	Sampling rate

Parameters

cutoffFrequency	Cutoff frequency.
gainDb	Gain in the passband
rippleDb	Permitted ripples in dB in the passband

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::LowShelf< FilterOrder, StateType >::setup (
            int reqOrder,
            double sampleRate,
            double cutoffFrequency,
            double gainDb,
            double rippleDb )  [inline]
```

Calculates the coefficients of the filter at specified order

Parameters

reqOrder	Actual order for the filter calculations
sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency.
gainDb	Gain in the passband
rippleDb	Permitted ripples in dB in the passband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::LowShelf< FilterOrder, StateType >::setupN (
            double cutoffFrequency,
            double gainDb,
            double rippleDb )  [inline]
```

Calculates the coefficients of the filter at the order FilterOrder

Parameters

cutoffFrequency	Normalised cutoff frequency (0..1/2)
gainDb	Gain in the passband
rippleDb	Permitted ripples in dB in the passband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevI::LowShelf< FilterOrder, StateType >::setupN (
            int reqOrder,
            double cutoffFrequency,
            double gainDb,
            double rippleDb )  [inline]
```

Calculates the coefficients of the filter at specified order

Parameters

reqOrder	Actual order for the filter calculations

Parameters

cutoffFrequency	Normalised cutoff frequency (0..1/2)
gainDb	Gain in the passband
rippleDb	Permitted ripples in dB in the passband

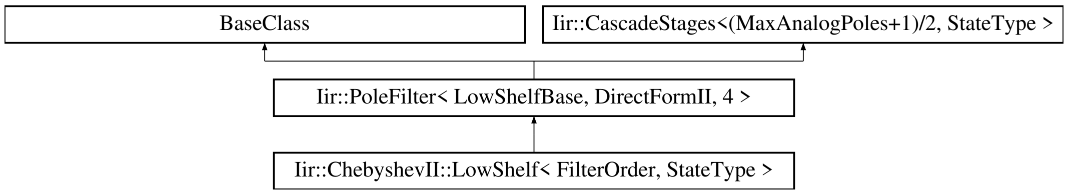
The documentation for this struct was generated from the following file:

- iir/ChebyshevI.h

7.67 Iir::ChebyshevII::LowShelf< FilterOrder, StateType > Struct Template Reference

`#include <ChebyshevII.h>`

Inheritance diagram for Iir::ChebyshevII::LowShelf< FilterOrder, StateType >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< LowShelfBase, DirectFormII, 4 >
Iir::ChebyshevII::LowShelf< FilterOrder, StateType >

Public Member Functions

- void setup (double sampleRate, double cutoffFrequency, double gainDb, double stopBandDb)
- void setup (int reqOrder, double sampleRate, double cutoffFrequency, double gainDb, double stopBandDb)
- void setupN (double cutoffFrequency, double gainDb, double stopBandDb)
- void setupN (int reqOrder, double cutoffFrequency, double gainDb, double stopBandDb)

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.67.1 Detailed Description

template<int FilterOrder = 4, class StateType = DirectFormII>
struct Iir::ChebyshevII::LowShelf< FilterOrder, StateType >

ChebyshevII low shelf filter. Specified gain in the passband and 0dB in the stopband.

Parameters

FilterOrder	Reserves memory for a filter of the order FilterOrder
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.67.2 Member Function Documentation

setup() `[1/2]`

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::LowShelf< FilterOrder, StateType >::setup (
            double sampleRate,
            double cutoffFrequency,
            double gainDb,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency.
gainDb	Gain of the passbard. The stopband has 0 dB gain.
stopBandDb	Permitted ripples in dB in the stopband

setup() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::LowShelf< FilterOrder, StateType >::setup (
            int reqOrder,
            double sampleRate,
            double cutoffFrequency,
            double gainDb,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

reqOrder	Requested order which can be less than the instantiated one
sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency
gainDb	Gain of the passbard. The stopband has 0 dB gain.
stopBandDb	Permitted ripples in dB in the stopband

setupN() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::LowShelf< FilterOrder, StateType >::setupN (
            double cutoffFrequency,
            double gainDb,
            double stopBandDb )  [inline]
```

Calculates the coefficients of the filter

Parameters

cutoffFrequency	Normalised cutoff frequency (0..1/2)
gainDb	Gain of the passbard. The stopband has 0 dB gain.
stopBandDb	Permitted ripples in dB in the stopband

setupN() [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void Iir::ChebyshevII::LowShelf< FilterOrder, StateType >::setupN (
            int reqOrder,
            double cutoffFrequency,
            double gainDb,
            double stopBandDb )  [inline]
```
Calculates the coefficients of the filter

Parameters

reqOrder	Requested order which can be less than the instantiated one
cutoffFrequency	Normalised cutoff frequency (0..1/2)
gainDb	Gain the passbard. The stopband has 0 dB gain.
stopBandDb	Permitted ripples in dB in the stopband

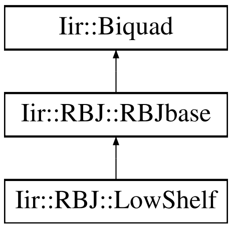
The documentation for this struct was generated from the following file:

- iir/ChebyshevII.h

7.68 Iir::RBJ::LowShelf Struct Reference

`#include <RBJ.h>`

Inheritance diagram for Iir::RBJ::LowShelf:



Public Member Functions

- void setupN (double cutoffFrequency, double gainDb, double shelfSlope=1)
- void setup (double sampleRate, double cutoffFrequency, double gainDb, double shelfSlope=1)

Public Member Functions inherited from Iir::RBJ::RBJbase

- template<typename Sample >
  Sample **filter** (Sample s)

  filter operation
- void **reset** ()

  resets the delay lines to zero
- const DirectFormI & **getState** ()

  gets the delay lines (=state) of the filter

Public Member Functions inherited from Iir::Biquad

- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const
- double getA0 () const
- double getA1 () const
- double getA2 () const
- double getB0 () const
- double getB1 () const
- double getB2 () const
- template<class StateType >
  double filter (double s, StateType &state) const
- void setCoefficients (double a0, double a1, double a2, double b0, double b1, double b2)

- void setOnePole (complex_t pole, complex_t zero)
- void setTwoPole (complex_t pole1, complex_t zero1, complex_t pole2, complex_t zero2)
- void setPoleZeroPair (const PoleZeroPair &pair)
- void setIdentity ()
- void applyScale (double scale)

7.68.1 Detailed Description

Low shelf: 0db in the stopband and gainDb in the passband.

7.68.2 Member Function Documentation

setup()

```
void Iir::RBJ::LowShelf::setup (
             double sampleRate,
             double cutoffFrequency,
             double gainDb,
             double shelfSlope = 1 )  [inline]
```

Calculates the coefficients

Parameters

sampleRate	Sampling rate
cutoffFrequency	Cutoff frequency
gainDb	Gain in the passband
shelfSlope	Slope between stop/passband. 1 = as steep as it can.

setupN()

```
void Iir::RBJ::LowShelf::setupN (
             double cutoffFrequency,
             double gainDb,
             double shelfSlope = 1 )
```

Calculates the coefficients

Parameters

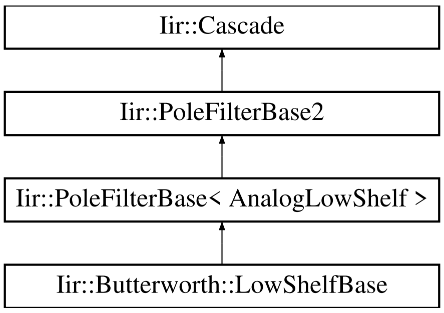
cutoffFrequency	Normalised cutoff frequency
gainDb	Gain in the passband
shelfSlope	Slope between stop/passband. 1 = as steep as it can.

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

- iir/RBJ.h
- iir/RBJ.cpp

7.69 Iir::Butterworth::LowShelfBase Struct Reference

Inheritance diagram for Iir::Butterworth::LowShelfBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowShelf >
Iir::Butterworth::LowShelfBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

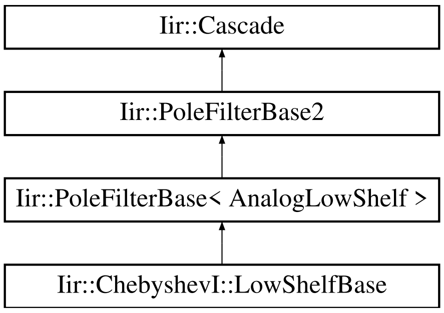
- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/Butterworth.h
- iir/Butterworth.cpp

7.70 Iir::ChebyshevI::LowShelfBase Struct Reference

Inheritance diagram for Iir::ChebyshevI::LowShelfBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowShelf >
Iir::ChebyshevI::LowShelfBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

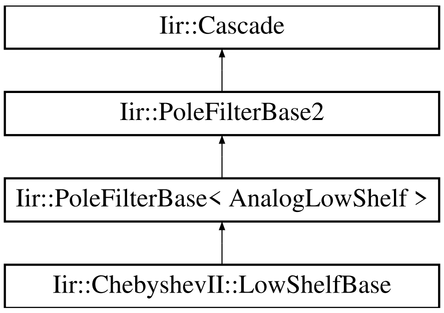
- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

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

- iir/ChebyshevI.h
- iir/ChebyshevI.cpp

7.71 Iir::ChebyshevII::LowShelfBase Struct Reference

Inheritance diagram for Iir::ChebyshevII::LowShelfBase:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowShelf >
Iir::ChebyshevII::LowShelfBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

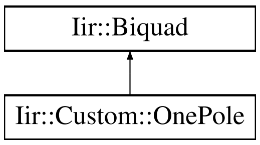
The documentation for this struct was generated from the following files:

- iir/ChebyshevII.h
- iir/ChebyshevII.cpp

7.72 Iir::Custom::OnePole Struct Reference

`#include <Custom.h>`

Inheritance diagram for Iir::Custom::OnePole:



Additional Inherited Members

Public Member Functions inherited from Iir::Biquad

- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const
- double getA0 () const
- double getA1 () const
- double getA2 () const
- double getB0 () const
- double getB1 () const
- double getB2 () const
- template<class StateType >
  double filter (double s, StateType &state) const
- void setCoefficients (double a0, double a1, double a2, double b0, double b1, double b2)
- void setOnePole (complex_t pole, complex_t zero)
- void setTwoPole (complex_t pole1, complex_t zero1, complex_t pole2, complex_t zero2)
- void setPoleZeroPair (const PoleZeroPair &pair)
- void setIdentity ()
- void applyScale (double scale)

7.72.1 Detailed Description

Setting up a filter with with one real pole, real zero and scale it by the scale factor

Parameters

scale	Scale the FIR coefficients by this factor
pole	Position of the pole on the real axis
zero	Position of the zero on the real axis

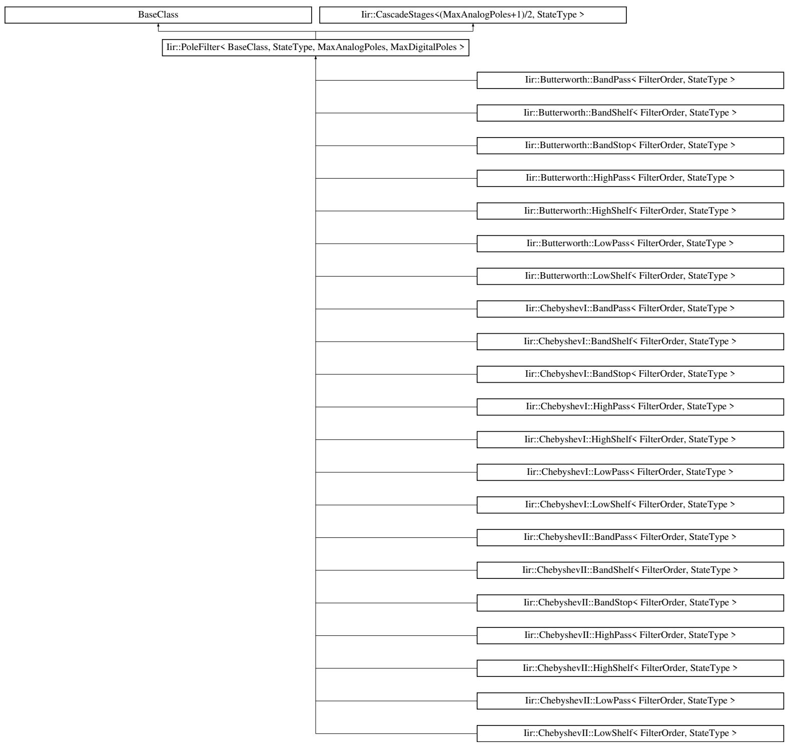
The documentation for this struct was generated from the following files:

- iir/Custom.h
- iir/Custom.cpp

7.73 Iir::PoleFilter< BaseClass, StateType, MaxAnalogPoles, MaxDigitalPoles > Struct Template Reference

`#include <PoleFilter.h>`

Inheritance diagram for Iir::PoleFilter< BaseClass, StateType, MaxAnalogPoles, MaxDigitalPoles >:


BaseClass
Iir::CascadeStages<(MaxAnalogPoles+1)/2, StateType >
Iir::PoleFilter< BaseClass, StateType, MaxAnalogPoles, MaxDigitalPoles >
Iir::Butterworth::BandPass< FilterOrder, StateType >
Iir::Butterworth::BandShelf< FilterOrder, StateType >
Iir::Butterworth::BandStop< FilterOrder, StateType >
Iir::Butterworth::HighPass< FilterOrder, StateType >
Iir::Butterworth::HighShelf< FilterOrder, StateType >
Iir::Butterworth::LowPass< FilterOrder, StateType >
Iir::Butterworth::LowShelf< FilterOrder, StateType >
Iir::ChebyshevI::BandPass< FilterOrder, StateType >
Iir::ChebyshevI::BandShelf< FilterOrder, StateType >
Iir::ChebyshevI::BandStop< FilterOrder, StateType >
Iir::ChebyshevI::HighPass< FilterOrder, StateType >
Iir::ChebyshevI::HighShelf< FilterOrder, StateType >
Iir::ChebyshevI::LowPass< FilterOrder, StateType >
Iir::ChebyshevI::LowShelf< FilterOrder, StateType >
Iir::ChebyshevII::BandPass< FilterOrder, StateType >
Iir::ChebyshevII::BandShelf< FilterOrder, StateType >
Iir::ChebyshevII::BandStop< FilterOrder, StateType >
Iir::ChebyshevII::HighPass< FilterOrder, StateType >
Iir::ChebyshevII::HighShelf< FilterOrder, StateType >
Iir::ChebyshevII::LowPass< FilterOrder, StateType >
Iir::ChebyshevII::LowShelf< FilterOrder, StateType >

Additional Inherited Members

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.73.1 Detailed Description

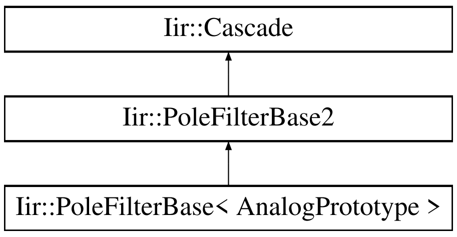
**template<class BaseClass, class StateType, int MaxAnalogPoles, int MaxDigitalPoles = MaxAnalogPoles>
struct Iir::PoleFilter< BaseClass, StateType, MaxAnalogPoles, MaxDigitalPoles >**

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

- iir/PoleFilter.h

7.74 Iir::PoleFilterBase< AnalogPrototype > Class Template Reference

```
#include <PoleFilter.h>
```
Inheritance diagram for Iir::PoleFilterBase< AnalogPrototype >:



Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

7.74.1 Detailed Description

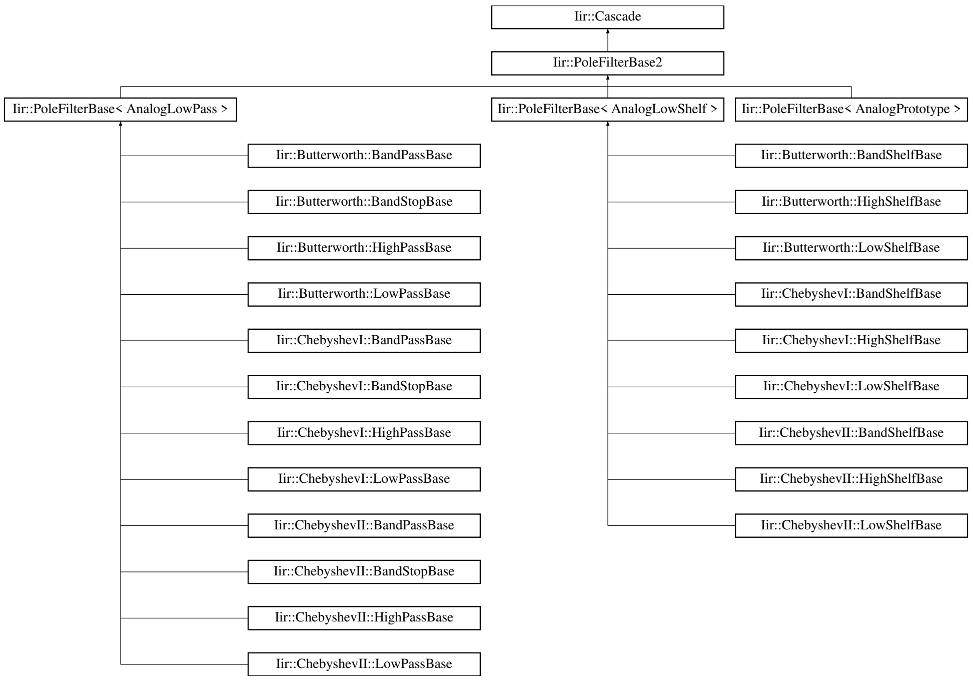
template<class AnalogPrototype>
class Iir::PoleFilterBase< AnalogPrototype >

Serves a container to hold the analog prototype and the digital pole/zero layout.
The documentation for this class was generated from the following file:

- iir/PoleFilter.h

7.75 Iir::PoleFilterBase2 Class Reference

```
#include <PoleFilter.h>
```
Inheritance diagram for Iir::PoleFilterBase2:


Iir::Cascade
Iir::PoleFilterBase2
Iir::PoleFilterBase< AnalogLowPass >
Iir::PoleFilterBase< AnalogLowShelf >
Iir::PoleFilterBase< AnalogPrototype >
Iir::Butterworth::BandPassBase
Iir::Butterworth::BandStopBase
Iir::Butterworth::HighPassBase
Iir::Butterworth::LowPassBase
Iir::ChebyshevI::BandPassBase
Iir::ChebyshevI::BandStopBase
Iir::ChebyshevI::HighPassBase
Iir::ChebyshevI::LowPassBase
Iir::ChebyshevII::BandPassBase
Iir::ChebyshevII::BandStopBase
Iir::ChebyshevII::HighPassBase
Iir::ChebyshevII::LowPassBase
Iir::Butterworth::BandShelfBase
Iir::Butterworth::HighShelfBase
Iir::Butterworth::LowShelfBase
Iir::ChebyshevI::BandShelfBase
Iir::ChebyshevI::HighShelfBase
Iir::ChebyshevI::LowShelfBase
Iir::ChebyshevII::BandShelfBase
Iir::ChebyshevII::HighShelfBase
Iir::ChebyshevII::LowShelfBase

Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

- int getNumStages () const
- const Biquad & operator[ ] (int index)
- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const

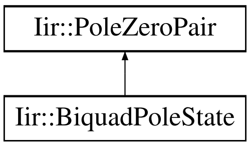
7.75.1 Detailed Description

Factored implementations to reduce template instantiations
The documentation for this class was generated from the following file:

- iir/PoleFilter.h

7.76 Iir::PoleZeroPair Struct Reference

`#include <Types.h>`
Inheritance diagram for Iir::PoleZeroPair:



7.76.1 Detailed Description

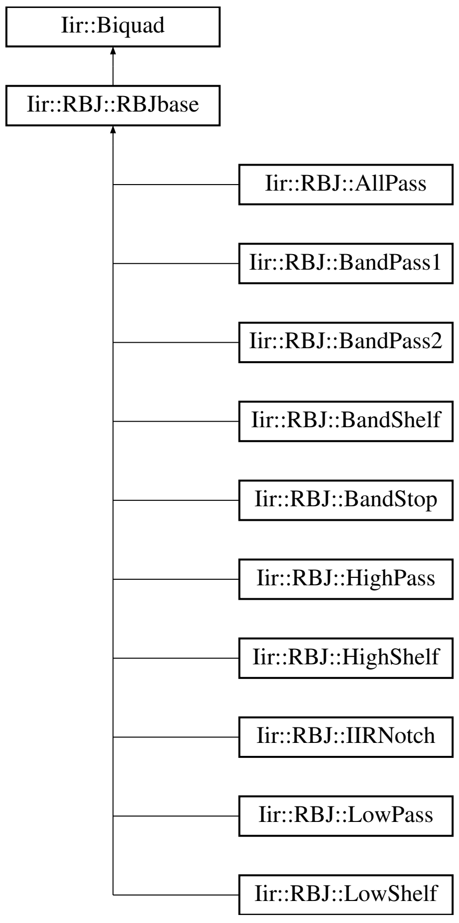
A pair of pole/zeros. This fits in a biquad (but is missing the gain)
The documentation for this struct was generated from the following file:

- iir/Types.h

7.77 Iir::RBJ::RBJbase Struct Reference

`#include <RBJ.h>`

Inheritance diagram for Iir::RBJ::RBJbase:



Public Member Functions

- template<typename Sample >
  Sample **filter** (Sample s)

  filter operation
- void **reset** ()

  resets the delay lines to zero
- const DirectFormI & **getState** ()

  gets the delay lines (=state) of the filter

Public Member Functions inherited from Iir::Biquad

- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const
- double getA0 () const
- double getA1 () const
- double getA2 () const
- double getB0 () const
- double getB1 () const
- double getB2 () const
- template<class StateType >
  double filter (double s, StateType &state) const

- void setCoefficients (double a0, double a1, double a2, double b0, double b1, double b2)
- void setOnePole (complex_t pole, complex_t zero)
- void setTwoPole (complex_t pole1, complex_t zero1, complex_t pole2, complex_t zero2)
- void setPoleZeroPair (const PoleZeroPair &pair)
- void setIdentity ()
- void applyScale (double scale)

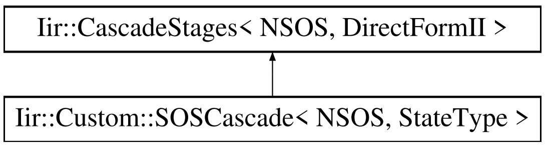
7.77.1 Detailed Description

The base class of all RBJ filters
The documentation for this struct was generated from the following file:

- iir/RBJ.h

7.78 Iir::Custom::SOSCascade< NSOS, StateType > Struct Template Reference

`#include <Custom.h>`
Inheritance diagram for Iir::Custom::SOSCascade< NSOS, StateType >:


Iir::CascadeStages< NSOS, DirectFormII >
Iir::Custom::SOSCascade< NSOS, StateType >

Public Member Functions

- SOSCascade ()=default
- SOSCascade (const double(&sosCoefficients)[NSOS][6])
- void setup (const double(&sosCoefficients)[NSOS][6])

Public Member Functions inherited from Iir::CascadeStages< MaxStages, StateType >

- void reset ()
- void setup (const double(&sosCoefficients)[MaxStages][6])
- template<typename Sample >
  Sample filter (const Sample in)
- const Cascade::Storage getCascadeStorage ()

7.78.1 Detailed Description

template<int NSOS, class StateType = DirectFormII>
struct Iir::Custom::SOSCascade< NSOS, StateType >

A custom cascade of 2nd order (SOS / biquads) filters.

Parameters

NSOS	The number of 2nd order filters / biquads.
StateType	The filter topology: DirectFormI, DirectFormII, ...

7.78.2 Constructor & Destructor Documentation

SOSCascade() `[1/2]`

```
template<int NSOS, class StateType = DirectFormII>
Iir::Custom::SOSCascade< NSOS, StateType >::SOSCascade ( )  [default]
```

Default constructor which creates a unity gain filter of NSOS biquads. Set the filter coefficients later with the setup() method.

SOSCascade() [2/2]

```
template<int NSOS, class StateType = DirectFormII>
Iir::Custom::SOSCascade< NSOS, StateType >::SOSCascade (
            const double(&) sosCoefficients[NSOS][6] )  [inline]
```

Python scipy.signal-friendly setting of coefficients. Initialises the coefficients of the whole chain of biquads / SOS. The argument is a 2D array where the 1st dimension holds an array of 2nd order biquad / SOS coefficients. The six SOS coefficients are ordered "Python" style with first the FIR coefficients (B) and then the IIR coefficients (A). The 2D const double array needs to have exactly the size [NSOS][6].

Parameters

sosCoefficients	2D array Python style sos[NSOS][6]. Indexing: 0-2: FIR-, 3-5: IIR-coefficients.

7.78.3 Member Function Documentation

setup()

```
template<int NSOS, class StateType = DirectFormII>
void Iir::Custom::SOSCascade< NSOS, StateType >::setup (
            const double(&) sosCoefficients[NSOS][6] )  [inline]
```

Python scipy.signal-friendly setting of coefficients. Sets the coefficients of the whole chain of biquads / SOS. The argument is a 2D array where the 1st dimension holds an array of 2nd order biquad / SOS coefficients. The six SOS coefficients are ordered "Python" style with first the FIR coefficients (B) and then the IIR coefficients (A). The 2D const double array needs to have exactly the size [NSOS][6].

Parameters

sosCoefficients	2D array Python style sos[NSOS][6]. Indexing: 0-2: FIR-, 3-5: IIR-coefficients.

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

- iir/Custom.h

7.79 Iir::Cascade::Storage Struct Reference

```
#include <Cascade.h>
```

7.79.1 Detailed Description

To return the array from a function and to set it. Transmits number of stages and the pointer to the array.
The documentation for this struct was generated from the following file:

- iir/Cascade.h

7.80 Iir::TransposedDirectFormII Class Reference

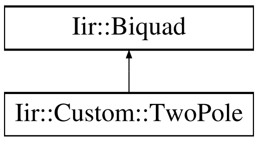
The documentation for this class was generated from the following file:

- iir/State.h

7.81 Iir::Custom::TwoPole Struct Reference

```
#include <Custom.h>
```

Inheritance diagram for Iir::Custom::TwoPole:



Additional Inherited Members

Public Member Functions inherited from Iir::Biquad

- complex_t response (double normalizedFrequency) const
- std::vector< PoleZeroPair > getPoleZeros () const
- double getA0 () const
- double getA1 () const
- double getA2 () const
- double getB0 () const
- double getB1 () const
- double getB2 () const
- template<class StateType >
  double filter (double s, StateType &state) const
- void setCoefficients (double a0, double a1, double a2, double b0, double b1, double b2)
- void setOnePole (complex_t pole, complex_t zero)
- void setTwoPole (complex_t pole1, complex_t zero1, complex_t pole2, complex_t zero2)
- void setPoleZeroPair (const PoleZeroPair &pair)
- void setIdentity ()
- void applyScale (double scale)

7.81.1 Detailed Description

Set a pole/zero pair in polar coordinates and scale the FIR filter coefficients

Parameters

poleRho	Radius of the pole
poleTheta	Angle of the pole
zeroRho	Radius of the zero
zeroTheta	Angle of the zero

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

- iir/Custom.h
- iir/Custom.cpp

8 File Documentation

8.1 Iir.h

```
#ifndef IIR_H
#define IIR_H

//
// Include this file in your application to get everything
//

#include "iir/Common.h"

#include "iir/Biquad.h"
#include "iir/Cascade.h"
#include "iir/PoleFilter.h"
#include "iir/State.h"
```

```

#include "iir/Butterworth.h"
#include "iir/ChebyshevI.h"
#include "iir/ChebyshevII.h"
#include "iir/Custom.h"
#include "iir/RBJ.h"

#endif
```

8.2 Biquad.h

```

#ifndef IIR1_BIQUAD_H
#define IIR1_BIQUAD_H

#include "Common.h"
#include "MathSupplement.h"
#include "Types.h"

namespace Iir {

        struct IIR_EXPORT BiquadPoleState;

/*
 * Holds coefficients for a second order Infinite Impulse Response
 * digital filter. This is the building block for all IIR filters.
 *
 */
        class IIR_EXPORT Biquad {
        public:

        Biquad() = default;

        complex_t response (double normalizedFrequency) const;

        std::vector<PoleZeroPair> getPoleZeros () const;

        double getA0 () const { return m_a0; }

        double getA1 () const { return m_a1*m_a0; }

        double getA2 () const { return m_a2*m_a0; }

        double getB0 () const { return m_b0*m_a0; }

        double getB1 () const { return m_b1*m_a0; }

        double getB2 () const { return m_b2*m_a0; }

        template <class StateType>
        inline double filter(double s, StateType& state) const
        {
                return state.filter(s, *this);
        }

        public:
        void setCoefficients (double a0, double a1, double a2,
                              double b0, double b1, double b2);

        void setOnePole (complex_t pole, complex_t zero);

        void setTwoPole (complex_t pole1, complex_t zero1,
                         complex_t pole2, complex_t zero2);

        void setPoleZeroPair (const PoleZeroPair& pair)
        {
                if (pair.isSinglePole ())
                        setOnePole (pair.poles.first, pair.zeros.first);
                else
                        setTwoPole (pair.poles.first, pair.zeros.first,
                                    pair.poles.second, pair.zeros.second);
        }

        void setPoleZeroForm (const BiquadPoleState& bps);

        void setIdentity ();

        void applyScale (double scale);

        public:
        double m_a0 = 1.0;
        double m_a1 = 0.0;
        double m_a2 = 0.0;
        double m_b1 = 0.0;
        double m_b2 = 0.0;
        double m_b0 = 1.0;
        };

//------------------------------------------------------------------------------


        struct IIR_EXPORT BiquadPoleState : PoleZeroPair
        {
                BiquadPoleState () = default;

                explicit BiquadPoleState (const Biquad& s);

                double gain = 1.0;
        };

}

#endif
```

8.3 Butterworth.h

```

#ifndef IIR1_BUTTERWORTH_H
#define IIR1_BUTTERWORTH_H

#include "Common.h"
#include "Cascade.h"
#include "PoleFilter.h"
#include "State.h"

namespace Iir {

namespace Butterworth {

class IIR_EXPORT AnalogLowPass : public LayoutBase
{
public:
        AnalogLowPass ();

        void design (const int numPoles);

private:
        int m_numPoles = 0;
};

//------------------------------------------------------------------------------

class IIR_EXPORT AnalogLowShelf : public LayoutBase
{
public:
        AnalogLowShelf ();

        void design (int numPoles, double gainDb);

private:
        int m_numPoles = 0;
        double m_gainDb = 0.0;
};

//------------------------------------------------------------------------------

struct IIR_EXPORT LowPassBase : PoleFilterBase <AnalogLowPass>
{
        void setup (int order,
                    double cutoffFrequency);
};

struct IIR_EXPORT HighPassBase : PoleFilterBase <AnalogLowPass>
{
        void setup (int order,
                    double cutoffFrequency);
};

struct IIR_EXPORT BandPassBase : PoleFilterBase <AnalogLowPass>
{
        void setup (int order,
                    double centerFrequency,
                    double widthFrequency);
};

struct IIR_EXPORT BandStopBase : PoleFilterBase <AnalogLowPass>
{
        void setup (int order,
                    double centerFrequency,
                    double widthFrequency);
};

```

```
struct IIR_EXPORT LowShelfBase : PoleFilterBase <AnalogLowShelf>
{
        void setup (int order,
                    double cutoffFrequency,
                    double gainDb);
};

struct IIR_EXPORT HighShelfBase : PoleFilterBase <AnalogLowShelf>
{
        void setup (int order,
                    double cutoffFrequency,
                    double gainDb);
};

struct IIR_EXPORT BandShelfBase : PoleFilterBase <AnalogLowShelf>
{
        void setup (int order,
                    double centerFrequency,
                    double widthFrequency,
                    double gainDb);
};

//------------------------------------------------------------------------------

//
// Filters for the user
//

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
struct LowPass : PoleFilter <LowPassBase, StateType, FilterOrder>
{
        void setup (double sampleRate,
                    double cutoffFrequency) {
                LowPassBase::setup (FilterOrder,
                                    cutoffFrequency / sampleRate);
        }

        void setup (int reqOrder,
                    double sampleRate,
                    double cutoffFrequency) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                LowPassBase::setup (reqOrder,
                                    cutoffFrequency / sampleRate);
        }


        void setupN(double cutoffFrequency) {
                LowPassBase::setup (FilterOrder,
                                    cutoffFrequency);
        }

        void setupN(int reqOrder,
                    double cutoffFrequency) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                LowPassBase::setup (reqOrder,
                                    cutoffFrequency);
        }
};

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
struct HighPass : PoleFilter <HighPassBase, StateType, FilterOrder>
{

        void setup (double sampleRate,
                    double cutoffFrequency) {
                HighPassBase::setup (FilterOrder,
                                     cutoffFrequency / sampleRate);
        }
        void setup (int reqOrder,
                    double sampleRate,
                    double cutoffFrequency) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                HighPassBase::setup (reqOrder,
                                     cutoffFrequency / sampleRate);
        }


        void setupN(double cutoffFrequency) {
                HighPassBase::setup (FilterOrder,
                                     cutoffFrequency);
        }
        void setupN(int reqOrder,
                    double cutoffFrequency) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                HighPassBase::setup (reqOrder,
                                     cutoffFrequency);
        }
```

```
};

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
struct BandPass : PoleFilter <BandPassBase, StateType, FilterOrder, FilterOrder*2>
{
        void setup (double sampleRate,
                    double centerFrequency,
                    double widthFrequency) {
                BandPassBase::setup(FilterOrder,
                                    centerFrequency / sampleRate,
                                    widthFrequency / sampleRate);
        }

        void setup (int reqOrder,
                    double sampleRate,
                    double centerFrequency,
                    double widthFrequency) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                BandPassBase::setup(reqOrder,
                                    centerFrequency / sampleRate,
                                    widthFrequency / sampleRate);
        }



        void setupN(double centerFrequency,
                    double widthFrequency) {
                BandPassBase::setup(FilterOrder,
                                    centerFrequency,
                                    widthFrequency);
        }

        void setupN(int reqOrder,
                    double centerFrequency,
                    double widthFrequency) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                BandPassBase::setup(reqOrder,
                                    centerFrequency,
                                    widthFrequency);
        }
};


template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
struct BandStop : PoleFilter <BandStopBase, StateType, FilterOrder, FilterOrder*2>
{
        void setup (double sampleRate,
                    double centerFrequency,
                    double widthFrequency) {
                BandStopBase::setup (FilterOrder,
                                     centerFrequency / sampleRate,
                                     widthFrequency / sampleRate);
        }

        void setup (int reqOrder,
                    double sampleRate,
                    double centerFrequency,
                    double widthFrequency) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                BandStopBase::setup (reqOrder,
                                     centerFrequency / sampleRate,
                                     widthFrequency / sampleRate);
        }



        void setupN(double centerFrequency,
                    double widthFrequency) {
                BandStopBase::setup (FilterOrder,
                                     centerFrequency,
                                     widthFrequency);
        }

        void setupN(int reqOrder,
                    double centerFrequency,
                    double widthFrequency) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                BandStopBase::setup (reqOrder,
                                     centerFrequency,
                                     widthFrequency);
        }

};

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
struct LowShelf : PoleFilter <LowShelfBase, StateType, FilterOrder>
{
        void setup (double sampleRate,
                    double cutoffFrequency,
                    double gainDb) {
                LowShelfBase::setup (FilterOrder,
                                     cutoffFrequency / sampleRate,
                                     gainDb);
        }

        void setup (int reqOrder,
                    double sampleRate,
                    double cutoffFrequency,
                    double gainDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                LowShelfBase::setup (reqOrder,
                                     cutoffFrequency / sampleRate,
                                     gainDb);
        }




        void setupN(double cutoffFrequency,
                    double gainDb) {
                LowShelfBase::setup (FilterOrder,
                                     cutoffFrequency,
                                     gainDb);
        }

        void setupN(int reqOrder,
                    double cutoffFrequency,
                    double gainDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                LowShelfBase::setup (reqOrder,
                                     cutoffFrequency,
                                     gainDb);
        }

};


template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
struct HighShelf : PoleFilter <HighShelfBase, StateType, FilterOrder>
{
        void setup (double sampleRate,
                    double cutoffFrequency,
                    double gainDb) {
                HighShelfBase::setup (FilterOrder,
                                      cutoffFrequency / sampleRate,
                                      gainDb);
        }

        void setup (int reqOrder,
                    double sampleRate,
                    double cutoffFrequency,
                    double gainDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                HighShelfBase::setup (reqOrder,
                                      cutoffFrequency / sampleRate,
                                      gainDb);
        }



        void setupN(double cutoffFrequency,
                    double gainDb) {
                HighShelfBase::setup (FilterOrder,
                                      cutoffFrequency,
                                      gainDb);
        }

        void setupN(int reqOrder,
                    double cutoffFrequency,
                    double gainDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                HighShelfBase::setup (reqOrder,
                                      cutoffFrequency,
                                      gainDb);
        }
};


template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
struct BandShelf : PoleFilter <BandShelfBase, StateType, FilterOrder, FilterOrder*2>
{
        void setup (double sampleRate,
                    double centerFrequency,
                    double widthFrequency,
                    double gainDb) {
                BandShelfBase::setup (FilterOrder,
                                      centerFrequency / sampleRate,
                                      widthFrequency / sampleRate,
                                      gainDb);
        }

        void setup (int reqOrder,
                    double sampleRate,
                    double centerFrequency,
                    double widthFrequency,
                    double gainDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                BandShelfBase::setup (reqOrder,
                                      centerFrequency / sampleRate,
                                      widthFrequency / sampleRate,
                                      gainDb);
        }



        void setupN(double centerFrequency,
                    double widthFrequency,
                    double gainDb) {
                BandShelfBase::setup (FilterOrder,
                                      centerFrequency,
                                      widthFrequency,
                                      gainDb);
        }

        void setupN(int reqOrder,
                    double centerFrequency,
                    double widthFrequency,
                    double gainDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                BandShelfBase::setup (reqOrder,
                                      centerFrequency,
                                      widthFrequency,
                                      gainDb);
        }
};

}

}

#endif

```

8.4 Cascade.h

```

#ifndef IIR1_CASCADE_H
#define IIR1_CASCADE_H

#include "Common.h"
#include "Biquad.h"
#include "Layout.h"
#include "MathSupplement.h"

namespace Iir {

        class IIR_EXPORT Cascade
        {
        public:

        Cascade () = default;

        struct IIR_EXPORT Storage
        {
            Storage() = delete;
            Storage(const int maxNumBiquads, Biquad* const biquadArray) : maxStages(maxNumBiquads),
stageArray(biquadArray) {}
            const int maxStages = 0;
            Biquad* const stageArray = nullptr;
        };

        int getNumStages () const
        {
                return m_numStages;
        }

        const Biquad& operator[] (int index)
        {
                if ((index < 0) || (index >= m_numStages))
                        throw_invalid_argument("Index out of bounds.");
                return m_stageArray[index];
        }

        complex_t response (double normalizedFrequency) const;

        std::vector<PoleZeroPair> getPoleZeros () const;

        void setCascadeStorage (const Storage& storage);

        void applyScale (double scale);

        void setLayout (const LayoutBase& proto);

        private:
        int m_numStages = 0;
        int m_maxStages = 0;
        Biquad* m_stageArray = nullptr;
        };


//------------------------------------------------------------------------------

        template <int MaxStages,class StateType>
        class CascadeStages {

        public:
        CascadeStages() = default;


        public:
        void reset ()
        {
                for (auto &state: m_states)
                        state.reset();
        }

        public:
        void setup (const double (&sosCoefficients)[MaxStages][6]) {
                for (int i = 0; i < MaxStages; i++) {
                        m_stages[i].setCoefficients(
                                sosCoefficients[i][3],
                                sosCoefficients[i][4],
                                sosCoefficients[i][5],
                                sosCoefficients[i][0],
                                sosCoefficients[i][1],
                                sosCoefficients[i][2]);
                }
        }

        public:
        template <typename Sample>
        inline Sample filter(const Sample in)
        {
                double out = in;
                StateType* state = m_states;
                for (const auto &stage: m_stages)
                        out = (state++)->filter(out, stage);
                return static_cast<Sample> (out);
        }

        const Cascade::Storage getCascadeStorage()
        {
            const Cascade::Storage s(MaxStages, m_stages);
            return s;
        }

        private:
        Biquad m_stages[MaxStages] = {};
        StateType m_states[MaxStages] = {};
        };

}

#endif
```

8.5 ChebyshevI.h

```

#ifndef IIR1_CHEBYSHEVI_H
#define IIR1_CHEBYSHEVI_H

#include "Common.h"
#include "Cascade.h"
#include "PoleFilter.h"
#include "State.h"

```

```
namespace Iir {

namespace ChebyshevI {

class IIR_EXPORT AnalogLowPass : public LayoutBase
{
public:
        AnalogLowPass ();

        void design (const int numPoles,
                     double rippleDb);

private:
        int m_numPoles = 0;
        double m_rippleDb = 0.0;
};

class IIR_EXPORT AnalogLowShelf : public LayoutBase
{
public:
        AnalogLowShelf ();

        void design (int numPoles,
                     double gainDb,
                     double rippleDb);

private:
        int m_numPoles = 0;
        double m_rippleDb = 0.0;
        double m_gainDb = 0.0;
};

//------------------------------------------------------------------------------

struct IIR_EXPORT LowPassBase : PoleFilterBase <AnalogLowPass>
{
  void setup (int order,
              double cutoffFrequency,
              double rippleDb);
};

struct IIR_EXPORT HighPassBase : PoleFilterBase <AnalogLowPass>
{
  void setup (int order,
              double cutoffFrequency,
              double rippleDb);
};

struct IIR_EXPORT BandPassBase : PoleFilterBase <AnalogLowPass>
{
  void setup (int order,
              double centerFrequency,
              double widthFrequency,
              double rippleDb);
};

struct IIR_EXPORT BandStopBase : PoleFilterBase <AnalogLowPass>
{
  void setup (int order,
              double centerFrequency,
              double widthFrequency,
              double rippleDb);
};

struct IIR_EXPORT LowShelfBase : PoleFilterBase <AnalogLowShelf>
{
  void setup (int order,
              double cutoffFrequency,
              double gainDb,
              double rippleDb);
};

struct IIR_EXPORT HighShelfBase : PoleFilterBase <AnalogLowShelf>
{
  void setup (int order,
              double cutoffFrequency,
              double gainDb,
              double rippleDb);
};

struct IIR_EXPORT BandShelfBase : PoleFilterBase <AnalogLowShelf>
{
  void setup (int order,
              double centerFrequency,
              double widthFrequency,
              double gainDb,
              double rippleDb);
};

//------------------------------------------------------------------------------

//
// Userland filters
//

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
        struct LowPass : PoleFilter <LowPassBase, StateType, FilterOrder>
        {
                void setup (double sampleRate,
                            double cutoffFrequency,
                            double rippleDb) {
                        LowPassBase::setup (FilterOrder,
                                            cutoffFrequency / sampleRate,
                                            rippleDb);
                }

                void setup (int reqOrder,
                            double sampleRate,
                            double cutoffFrequency,
                            double rippleDb) {
                        if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                        LowPassBase::setup (reqOrder,
                                            cutoffFrequency / sampleRate,
                                            rippleDb);
                }



                void setupN(double cutoffFrequency,
                            double rippleDb) {
                        LowPassBase::setup (FilterOrder,
                                            cutoffFrequency,
                                            rippleDb);
                }

                void setupN(int reqOrder,
                            double cutoffFrequency,
                            double rippleDb) {
                        if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                        LowPassBase::setup (reqOrder,
                                            cutoffFrequency,
                                            rippleDb);
                }
};

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
        struct HighPass : PoleFilter <HighPassBase, StateType, FilterOrder>
        {
                void setup (double sampleRate,
                            double cutoffFrequency,
                            double rippleDb) {
                        HighPassBase::setup (FilterOrder,
                                             cutoffFrequency / sampleRate,
                                             rippleDb);
                }

                void setup (int reqOrder,
                            double sampleRate,
                            double cutoffFrequency,
                            double rippleDb) {
                        if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                        HighPassBase::setup (reqOrder,
                                             cutoffFrequency / sampleRate,
                                             rippleDb);
                }



                void setupN(double cutoffFrequency,
                            double rippleDb) {
                        HighPassBase::setup (FilterOrder,
                                             cutoffFrequency,
                                             rippleDb);
                }

                void setupN(int reqOrder,
                            double cutoffFrequency,
                            double rippleDb) {
                        if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                        HighPassBase::setup (reqOrder,
                                             cutoffFrequency,
                                             rippleDb);
                }
};
```

```

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
        struct BandPass : PoleFilter <BandPassBase, StateType, FilterOrder, FilterOrder*2>
        {
                void setup (double sampleRate,
                            double centerFrequency,
                            double widthFrequency,
                            double rippleDb) {
                        BandPassBase::setup (FilterOrder,
                               centerFrequency / sampleRate,
                               widthFrequency / sampleRate,
                               rippleDb);
                }

                void setup (int reqOrder,
                            double sampleRate,
                            double centerFrequency,
                            double widthFrequency,
                            double rippleDb) {
                        if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                        BandPassBase::setup (reqOrder,
                               centerFrequency / sampleRate,
                               widthFrequency / sampleRate,
                               rippleDb);
                }



                void setupN(double centerFrequency,
                            double widthFrequency,
                            double rippleDb) {
                        BandPassBase::setup (FilterOrder,
                               centerFrequency,
                               widthFrequency,
                               rippleDb);
                }

                void setupN(int reqOrder,
                            double centerFrequency,
                            double widthFrequency,
                            double rippleDb) {
                        if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                        BandPassBase::setup (reqOrder,
                               centerFrequency,
                               widthFrequency,
                               rippleDb);
                }
};

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
        struct BandStop : PoleFilter <BandStopBase, StateType, FilterOrder, FilterOrder*2>
        {
                void setup (double sampleRate,
                            double centerFrequency,
                            double widthFrequency,
                            double rippleDb) {
                        BandStopBase::setup (FilterOrder,
                                             centerFrequency / sampleRate,
                                             widthFrequency / sampleRate,
                                             rippleDb);
                }

                void setup (int reqOrder,
                            double sampleRate,
                            double centerFrequency,
                            double widthFrequency,
                            double rippleDb) {
                        if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                        BandStopBase::setup (reqOrder,
                                             centerFrequency / sampleRate,
                                             widthFrequency / sampleRate,
                                             rippleDb);
                }



                void setupN(double centerFrequency,
                            double widthFrequency,
                            double rippleDb) {
                        BandStopBase::setup (FilterOrder,
                                             centerFrequency,
                                             widthFrequency,
                                             rippleDb);
                }

                void setupN(int reqOrder,
                            double centerFrequency,
                                double widthFrequency,
                                double rippleDb) {
                        if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                        BandStopBase::setup (reqOrder,
                                             centerFrequency,
                                             widthFrequency,
                                             rippleDb);
                }

};

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
        struct LowShelf : PoleFilter <LowShelfBase, StateType, FilterOrder>
        {
                void setup (double sampleRate,
                            double cutoffFrequency,
                            double gainDb,
                            double rippleDb) {
                        LowShelfBase::setup (FilterOrder,
                                             cutoffFrequency / sampleRate,
                                             gainDb,
                                             rippleDb);
                }

                void setup (int reqOrder,
                            double sampleRate,
                            double cutoffFrequency,
                            double gainDb,
                            double rippleDb) {
                        if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                        LowShelfBase::setup (reqOrder,
                                             cutoffFrequency / sampleRate,
                                             gainDb,
                                             rippleDb);
                }



                void setupN(double cutoffFrequency,
                            double gainDb,
                            double rippleDb) {
                        LowShelfBase::setup (FilterOrder,
                                             cutoffFrequency,
                                             gainDb,
                                             rippleDb);
                }

                void setupN(int reqOrder,
                            double cutoffFrequency,
                            double gainDb,
                            double rippleDb) {
                        if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                        LowShelfBase::setup (reqOrder,
                                             cutoffFrequency,
                                             gainDb,
                                             rippleDb);
                }
};

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
        struct HighShelf : PoleFilter <HighShelfBase, StateType, FilterOrder>
        {
                void setup (double sampleRate,
                            double cutoffFrequency,
                            double gainDb,
                            double rippleDb) {
                        HighShelfBase::setup (FilterOrder,
                                cutoffFrequency / sampleRate,
                                gainDb,
                                rippleDb);
                }

                void setup (int reqOrder,
                            double sampleRate,
                            double cutoffFrequency,
                            double gainDb,
                            double rippleDb) {
                        if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                        HighShelfBase::setup (reqOrder,
                                cutoffFrequency / sampleRate,
                                gainDb,
                                rippleDb);
                }




                void setupN(double cutoffFrequency,
                        double gainDb,
                        double rippleDb) {
                    HighShelfBase::setup (FilterOrder,
                        cutoffFrequency,
                        gainDb,
                        rippleDb);
                }

                void setupN(int reqOrder,
                        double cutoffFrequency,
                        double gainDb,
                        double rippleDb) {
                    if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                    HighShelfBase::setup (reqOrder,
                        cutoffFrequency,
                        gainDb,
                        rippleDb);
                }

};

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
        struct BandShelf : PoleFilter <BandShelfBase, StateType, FilterOrder, FilterOrder*2>
        {
                void setup (double sampleRate,
                        double centerFrequency,
                        double widthFrequency,
                        double gainDb,
                        double rippleDb) {
                    BandShelfBase::setup (FilterOrder,
                                      centerFrequency / sampleRate,
                                      widthFrequency / sampleRate,
                                      gainDb,
                                      rippleDb);

                }

                void setup (int reqOrder,
                        double sampleRate,
                        double centerFrequency,
                        double widthFrequency,
                        double gainDb,
                        double rippleDb) {
                    if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                    BandShelfBase::setup (reqOrder,
                                      centerFrequency / sampleRate,
                                      widthFrequency / sampleRate,
                                      gainDb,
                                      rippleDb);

                }




                void setupN(double centerFrequency,
                        double widthFrequency,
                        double gainDb,
                        double rippleDb) {
                    BandShelfBase::setup (FilterOrder,
                                      centerFrequency,
                                      widthFrequency,
                                      gainDb,
                                      rippleDb);

                }

                void setupN(int reqOrder,
                        double centerFrequency,
                        double widthFrequency,
                        double gainDb,
                        double rippleDb) {
                    if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                    BandShelfBase::setup (reqOrder,
                                      centerFrequency,
                                      widthFrequency,
                                      gainDb,
                                      rippleDb);

                }

        };

}

}
```

```

#endif
```

8.6 ChebyshevII.h

```

#ifndef IIR1_CHEBYSHEVII_H
#define IIR1_CHEBYSHEVII_H

#include "Common.h"
#include "Cascade.h"
#include "PoleFilter.h"
#include "State.h"

namespace Iir {

namespace ChebyshevII {

class IIR_EXPORT AnalogLowPass : public LayoutBase
{
public:
        AnalogLowPass ();

        void design (const int numPoles,
                     double stopBandDb);

private:
        int m_numPoles = 0;
        double m_stopBandDb = 0.0;
};


class IIR_EXPORT AnalogLowShelf : public LayoutBase
{
public:
        AnalogLowShelf ();

        void design (int numPoles,
                     double gainDb,
                     double stopBandDb);

private:
        int m_numPoles = 0;
        double m_stopBandDb = 0.0;
        double m_gainDb = 0.0;
};

//------------------------------------------------------------------------------

struct IIR_EXPORT LowPassBase : PoleFilterBase <AnalogLowPass>
{
        void setup (int order,
                    double cutoffFrequency,
                    double stopBandDb);
};

struct IIR_EXPORT HighPassBase : PoleFilterBase <AnalogLowPass>
{
        void setup (int order,
                    double cutoffFrequency,
                    double stopBandDb);
};

struct IIR_EXPORT BandPassBase : PoleFilterBase <AnalogLowPass>
{
        void setup (int order,
                    double centerFrequency,
                    double widthFrequency,
                    double stopBandDb);
};

struct IIR_EXPORT BandStopBase : PoleFilterBase <AnalogLowPass>
{
        void setup (int order,
                    double centerFrequency,
                    double widthFrequency,
                    double stopBandDb);
};

struct IIR_EXPORT LowShelfBase : PoleFilterBase <AnalogLowShelf>
{
        void setup (int order,
                    double cutoffFrequency,
                    double gainDb,
                    double stopBandDb);
};
```

```

struct IIR_EXPORT HighShelfBase : PoleFilterBase <AnalogLowShelf>
{
        void setup (int order,
                    double cutoffFrequency,
                    double gainDb,
                    double stopBandDb);
};

struct IIR_EXPORT BandShelfBase : PoleFilterBase <AnalogLowShelf>
{
        void setup (int order,
                    double centerFrequency,
                    double widthFrequency,
                    double gainDb,
                    double stopBandDb);
};

//------------------------------------------------------------------------------

//
// Userland filters
//

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
struct LowPass : PoleFilter <LowPassBase, StateType, FilterOrder>
{
        void setup (double sampleRate,
                    double cutoffFrequency,
                    double stopBandDb) {
                LowPassBase::setup (FilterOrder,
                                    cutoffFrequency / sampleRate,
                                    stopBandDb);
        }

        void setup (int reqOrder,
                    double sampleRate,
                    double cutoffFrequency,
                    double stopBandDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                LowPassBase::setup (reqOrder,
                                    cutoffFrequency / sampleRate,
                                    stopBandDb);
        }





        void setupN(double cutoffFrequency,
                    double stopBandDb) {
                LowPassBase::setup (FilterOrder,
                                    cutoffFrequency,
                                    stopBandDb);
        }

        void setupN(int reqOrder,
                    double cutoffFrequency,
                    double stopBandDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                LowPassBase::setup (reqOrder,
                                    cutoffFrequency,
                                    stopBandDb);
        }

};

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
struct HighPass : PoleFilter <HighPassBase, StateType, FilterOrder>
{
        void setup (double sampleRate,
                    double cutoffFrequency,
                    double stopBandDb) {
                HighPassBase::setup (FilterOrder,
                                     cutoffFrequency / sampleRate,
                                     stopBandDb);
        }

        void setup (int reqOrder,
                    double sampleRate,
                    double cutoffFrequency,
                    double stopBandDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                HighPassBase::setup (reqOrder,
                                     cutoffFrequency / sampleRate,
                                     stopBandDb);
        }




        void setupN(double cutoffFrequency,
                    double stopBandDb) {
                HighPassBase::setup (FilterOrder,
                                     cutoffFrequency,
                                     stopBandDb);
        }

        void setupN(int reqOrder,
                    double cutoffFrequency,
                    double stopBandDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                HighPassBase::setup (reqOrder,
                                     cutoffFrequency,
                                     stopBandDb);
        }

};

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
struct BandPass : PoleFilter <BandPassBase, StateType, FilterOrder, FilterOrder*2>
{
        void setup (double sampleRate,
                    double centerFrequency,
                    double widthFrequency,
                    double stopBandDb) {
                BandPassBase::setup (FilterOrder,
                                     centerFrequency / sampleRate,
                                     widthFrequency / sampleRate,
                                     stopBandDb);
        }

        void setup (int reqOrder,
                    double sampleRate,
                    double centerFrequency,
                    double widthFrequency,
                    double stopBandDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                BandPassBase::setup (reqOrder,
                                     centerFrequency / sampleRate,
                                     widthFrequency / sampleRate,
                                     stopBandDb);
        }




        void setupN(double centerFrequency,
                    double widthFrequency,
                    double stopBandDb) {
                BandPassBase::setup (FilterOrder,
                                     centerFrequency,
                                     widthFrequency,
                                     stopBandDb);
        }

        void setupN(int reqOrder,
                    double centerFrequency,
                    double widthFrequency,
                    double stopBandDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                BandPassBase::setup (reqOrder,
                                     centerFrequency,
                                     widthFrequency,
                                     stopBandDb);
        }
};

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
struct BandStop : PoleFilter <BandStopBase, StateType, FilterOrder, FilterOrder*2>
{
        void setup (double sampleRate,
                    double centerFrequency,
                    double widthFrequency,
                    double stopBandDb) {
                BandStopBase::setup (FilterOrder,
                                     centerFrequency / sampleRate,
                                     widthFrequency / sampleRate,
                                     stopBandDb);
        }

        void setup (int reqOrder,
                    double sampleRate,
                    double centerFrequency,
                        double widthFrequency,
                        double stopBandDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                BandStopBase::setup (reqOrder,
                                     centerFrequency / sampleRate,
                                     widthFrequency / sampleRate,
                                     stopBandDb);
        }




        void setupN(double centerFrequency,
                        double widthFrequency,
                        double stopBandDb) {
                BandStopBase::setup (FilterOrder,
                                     centerFrequency,
                                     widthFrequency,
                                     stopBandDb);
        }

        void setupN(int reqOrder,
                        double centerFrequency,
                        double widthFrequency,
                        double stopBandDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                BandStopBase::setup (reqOrder,
                                     centerFrequency,
                                     widthFrequency,
                                     stopBandDb);
        }
};

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
struct LowShelf : PoleFilter <LowShelfBase, StateType, FilterOrder>
{
        void setup (double sampleRate,
                        double cutoffFrequency,
                        double gainDb,
                        double stopBandDb) {
                LowShelfBase::setup (FilterOrder,
                                     cutoffFrequency / sampleRate,
                                     gainDb,
                                     stopBandDb);
        }

        void setup (int reqOrder,
                        double sampleRate,
                        double cutoffFrequency,
                        double gainDb,
                        double stopBandDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                LowShelfBase::setup (reqOrder,
                                     cutoffFrequency / sampleRate,
                                     gainDb,
                                     stopBandDb);
        }





        void setupN(double cutoffFrequency,
                        double gainDb,
                        double stopBandDb) {
                LowShelfBase::setup (FilterOrder,
                                     cutoffFrequency,
                                     gainDb,
                                     stopBandDb);
        }

        void setupN(int reqOrder,
                        double cutoffFrequency,
                        double gainDb,
                        double stopBandDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                LowShelfBase::setup (reqOrder,
                                     cutoffFrequency,
                                     gainDb,
                                     stopBandDb);
        }

};

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
struct HighShelf : PoleFilter <HighShelfBase, StateType, FilterOrder>
{
        void setup (double sampleRate,
                        double cutoffFrequency,
                        double gainDb,
                        double stopBandDb) {
                HighShelfBase::setup (FilterOrder,
                                      cutoffFrequency / sampleRate,
                                      gainDb,
                                      stopBandDb);
        }

        void setup (int reqOrder,
                        double sampleRate,
                        double cutoffFrequency,
                        double gainDb,
                        double stopBandDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                HighShelfBase::setup (reqOrder,
                                      cutoffFrequency / sampleRate,
                                      gainDb,
                                      stopBandDb);
        }






        void setupN(double cutoffFrequency,
                        double gainDb,
                        double stopBandDb) {
                HighShelfBase::setup (FilterOrder,
                                      cutoffFrequency,
                                      gainDb,
                                      stopBandDb);
        }

        void setupN(int reqOrder,
                        double cutoffFrequency,
                        double gainDb,
                        double stopBandDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                HighShelfBase::setup (reqOrder,
                                      cutoffFrequency,
                                      gainDb,
                                      stopBandDb);
        }

};

template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
struct BandShelf : PoleFilter <BandShelfBase, StateType, FilterOrder, FilterOrder*2>
{
        void setup (double sampleRate,
                        double centerFrequency,
                        double widthFrequency,
                        double gainDb,
                        double stopBandDb) {
                BandShelfBase::setup (FilterOrder,
                                      centerFrequency / sampleRate,
                                      widthFrequency / sampleRate,
                                      gainDb,
                                      stopBandDb);
        }


        void setup (int reqOrder,
                        double sampleRate,
                        double centerFrequency,
                        double widthFrequency,
                        double gainDb,
                        double stopBandDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                BandShelfBase::setup (reqOrder,
                                      centerFrequency / sampleRate,
                                      widthFrequency / sampleRate,
                                      gainDb,
                                      stopBandDb);
        }







        void setupN(double centerFrequency,
                        double widthFrequency,
                    double gainDb,
                    double stopBandDb) {
                BandShelfBase::setup (FilterOrder,
                                      centerFrequency,
                                      widthFrequency,
                                      gainDb,
                                      stopBandDb);
        }


        void setupN(int reqOrder,
                    double centerFrequency,
                    double widthFrequency,
                    double gainDb,
                    double stopBandDb) {
                if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
                BandShelfBase::setup (reqOrder,
                                      centerFrequency,
                                      widthFrequency,
                                      gainDb,
                                      stopBandDb);
        }


};

}

}

#endif
```

8.7 Common.h

```

#ifndef IIR1_COMMON_H
#define IIR1_COMMON_H

//
// This must be the first file included in every DspFilters header and source
//

#ifdef _MSC_VER
#  pragma warning (disable: 4100)
#  ifndef _CRT_SECURE_NO_WARNINGS
#    define _CRT_SECURE_NO_WARNINGS
#  endif
#endif

// This exports the classes/structures to the windows DLL
#if defined(_WIN32) && defined(iir_EXPORTS)
#define IIR_EXPORT __declspec( dllexport )
#else
#define IIR_EXPORT
#endif

#include <stdlib.h>

#include <cassert>
#include <cfloat>
#include <cmath>
#include <complex>
#include <cstring>
#include <string>
#include <limits>
#include <vector>
#include <stdexcept> // for invalid_argument

static const char orderTooHigh[] = "Requested order is too high. Provide a higher order for the template.";

#define DEFAULT_FILTER_ORDER 4

inline void throw_invalid_argument(const char* msg) {

#ifndef IIR1_NO_EXCEPTIONS
    throw std::invalid_argument(msg);
#else
    (void) msg; // Discard parameter
    abort();
#endif

}

#endif
```

8.8 Custom.h

```
#ifndef IIR1_CUSTOM_H
#define IIR1_CUSTOM_H

#include "Common.h"
#include "Biquad.h"
#include "Cascade.h"
#include "PoleFilter.h"
#include "State.h"


namespace Iir {

namespace Custom {

struct OnePole : public Biquad
{
        void setup (double scale,
                    double pole,
                    double zero);
};

struct TwoPole : public Biquad
{
        void setup (double scale,
                    double poleRho,
                    double poleTheta,
                    double zeroRho,
                    double zeroTheta);
};

template <int NSOS, class StateType = DEFAULT_STATE>
struct SOSCascade : CascadeStages<NSOS,StateType>
{
        SOSCascade() = default;
        SOSCascade(const double (&sosCoefficients)[NSOS][6]) {
                CascadeStages<NSOS,StateType>::setup(sosCoefficients);
        }
        void setup (const double (&sosCoefficients)[NSOS][6]) {
                CascadeStages<NSOS,StateType>::setup(sosCoefficients);
        }
};

}

}

#endif
```

8.9 Layout.h

```
#ifndef IIR1_LAYOUT_H
#define IIR1_LAYOUT_H

#include "Common.h"
#include "MathSupplement.h"

namespace Iir {

        static const char errPoleisNaN[] = "Pole to add is NaN.";
        static const char errZeroisNaN[] = "Zero to add is NaN.";

        static const char errCantAdd2ndOrder[] = "Can't add 2nd order after a 1st order filter.";

        static const char errPolesNotComplexConj[] = "Poles not complex conjugate.";
        static const char errZerosNotComplexConj[] = "Zeros not complex conjugate.";

        static const char pairIndexOutOfBounds[] = "Pair index out of bounds.";

        class IIR_EXPORT LayoutBase
        {
        public:
                LayoutBase ()
                        : m_numPoles (0)
                        , m_maxPoles (0)
                        , m_pair (nullptr)
                {
                }

                LayoutBase (int maxPoles, PoleZeroPair* pairs)
                        : m_numPoles (0)
                        , m_maxPoles (maxPoles)
                        , m_pair (pairs)
                {
                }

                void setStorage (const LayoutBase& other)
                {
                        m_numPoles = 0;
                        m_maxPoles = other.m_maxPoles;
                        m_pair = other.m_pair;
                }

                void reset ()
                {
                        m_numPoles = 0;
                }

                int getNumPoles () const
                {
                        return m_numPoles;
                }

                int getMaxPoles () const
                {
                        return m_maxPoles;
                }

                void add (const complex_t& pole, const complex_t& zero)
                {
                        if (m_numPoles&1)
                                throw_invalid_argument(errCantAdd2ndOrder);
                        if (Iir::is_nan(pole))
                                throw_invalid_argument(errPoleisNaN);
                        if (Iir::is_nan(zero))
                                throw_invalid_argument(errZeroisNaN);
                        m_pair[m_numPoles/2] = PoleZeroPair (pole, zero);
                        ++m_numPoles;
                }

                void addPoleZeroConjugatePairs (const complex_t& pole,
                                                const complex_t& zero)
                {
                        if (m_numPoles&1)
                                throw_invalid_argument(errCantAdd2ndOrder);
                        if (Iir::is_nan(pole))
                                throw_invalid_argument(errPoleisNaN);
                        if (Iir::is_nan(zero))
                                throw_invalid_argument(errZeroisNaN);
                        m_pair[m_numPoles/2] = PoleZeroPair (
                                pole, zero, std::conj (pole), std::conj (zero));
                        m_numPoles += 2;
                }

                void add (const ComplexPair& poles, const ComplexPair& zeros)
                {
                        if (m_numPoles&1)
                                throw_invalid_argument(errCantAdd2ndOrder);
                        if (!poles.isMatchedPair ())
                                throw_invalid_argument(errPolesNotComplexConj);
                        if (!zeros.isMatchedPair ())
                                throw_invalid_argument(errZerosNotComplexConj);
                        m_pair[m_numPoles/2] = PoleZeroPair (poles.first, zeros.first,
                                                             poles.second, zeros.second);
                        m_numPoles += 2;
                }

                const PoleZeroPair& getPair (int pairIndex) const
                {
                        if ((pairIndex < 0) || (pairIndex >= (m_numPoles+1)/2))
                                throw_invalid_argument(pairIndexOutOfBounds);
                        return m_pair[pairIndex];
                }

                const PoleZeroPair& operator[] (int pairIndex) const
                {
                        return getPair (pairIndex);
                }

                double getNormalW () const
                {
                        return m_normalW;
                }

                double getNormalGain () const
                {
                        return m_normalGain;
                }

                void setNormal (double w, double g)
                {
                        m_normalW = w;
                        m_normalGain = g;
                }

        private:
                int m_numPoles = 0;
                int m_maxPoles = 0;
                PoleZeroPair* m_pair = nullptr;
                double m_normalW = 0;
                double m_normalGain = 1;
        };

//------------------------------------------------------------------------------

        template <int MaxPoles>
                class Layout
        {
        public:
                operator LayoutBase ()
                {
                        return LayoutBase (MaxPoles, m_pairs);
                }

        private:
                PoleZeroPair m_pairs[(MaxPoles+1)/2] = {};
        };

}

#endif
```

8.10 MathSupplement.h

```

#ifndef IIR1_MATHSUPPLEMENT_H
#define IIR1_MATHSUPPLEMENT_H

#include "Common.h"

#include<complex>

#ifdef _MSC_VER
 // Under Unix these have already default instantiations but not under Vis Studio
template class IIR_EXPORT std::complex<double>;
template class IIR_EXPORT std::complex<float>;
#endif

namespace Iir {

const double doublePi   =3.1415926535897932384626433832795028841971;
const double doublePi_2 =1.5707963267948966192313216916397514420986;
const double doubleLn2  =0.69314718055994530941723212145818;
const double doubleLn10 =2.3025850929940456840179914546844;

typedef std::complex<double> complex_t;
typedef std::pair<complex_t, complex_t> complex_pair_t;

inline const complex_t infinity()
{
  return complex_t (std::numeric_limits<double>::infinity());
}

template <typename Ty, typename To>
inline std::complex<Ty> addmul (const std::complex<Ty>& c,
                                Ty v,
                                const std::complex<To>& c1)
{
  return std::complex <Ty> (
    c.real() + v * c1.real(), c.imag() + v * c1.imag());
}

template <typename Ty>
inline Ty asinh (Ty x)
{
  return log (x + std::sqrt (x * x + 1 ));
}

template <typename Ty>
inline bool is_nan (Ty v)
{
  return !(v == v);
}

template <>
inline bool is_nan<complex_t> (complex_t v)
{
  return Iir::is_nan (v.real()) || Iir::is_nan (v.imag());
}

}

#endif
```

8.11 PoleFilter.h

```

#ifndef IIR1_POLEFILTER_H
#define IIR1_POLEFILTER_H

#include "Common.h"
#include "MathSupplement.h"
#include "Cascade.h"
#include "State.h"

// Purely for debugging...
#include <iostream>

namespace Iir {

/***
 * Base for filters designed via algorithmic placement of poles and zeros.
 *
 * Typically, the filter is first designed as a half-band low pass or
 * low shelf analog filter (s-plane). Then, using a transformation such
 * as the ones from Constantinides, the poles and zeros of the analog filter
 * are calculated in the z-plane.
 *
 ***/

        class IIR_EXPORT PoleFilterBase2 : public Cascade
        {
        public:
                // This gets the poles/zeros directly from the digital
                // prototype. It is used to double check the correctness
                // of the recovery of pole/zeros from biquad coefficients.
                //
                // It can also be used to accelerate the interpolation
                // of pole/zeros for parameter modulation, since a pole
                // filter already has them calculated

                PoleFilterBase2() = default;

                std::vector<PoleZeroPair> getPoleZeros () const
                {
                        std::vector<PoleZeroPair> vpz;
                        const int pairs = (m_digitalProto.getNumPoles () + 1) / 2;
                        for (int i = 0; i < pairs; ++i)
                                vpz.push_back (m_digitalProto[i]);
                        return vpz;
                }

        protected:
                LayoutBase m_digitalProto = {};
        };


        template <class AnalogPrototype>
        class PoleFilterBase : public PoleFilterBase2
        {
        protected:
                void setPrototypeStorage (const LayoutBase& analogStorage,
                                          const LayoutBase& digitalStorage)
                {
                        m_analogProto.setStorage (analogStorage);
                        m_digitalProto = digitalStorage;
                }

        protected:
                AnalogPrototype m_analogProto = {};
        };

//------------------------------------------------------------------------------

        template <class BaseClass,
                  class StateType,
                  int MaxAnalogPoles,
                  int MaxDigitalPoles = MaxAnalogPoles>
        struct PoleFilter : BaseClass
                , CascadeStages <(MaxDigitalPoles + 1) / 2 , StateType>
        {
        public:
                PoleFilter ()
                        {
                                // This glues together the factored base classes
                                // with the templatized storage classes.
                                BaseClass::setCascadeStorage (this->getCascadeStorage());
                                BaseClass::setPrototypeStorage (m_analogStorage, m_digitalStorage);
                                CascadeStages<(MaxDigitalPoles + 1) / 2 , StateType>::reset();
                        }

                PoleFilter(const PoleFilter&) = default;

                PoleFilter& operator=(const PoleFilter&)
                        {
                                // Reset the filter state when copied for now
                                CascadeStages<(MaxDigitalPoles + 1) / 2 , StateType>::reset();
                                return *this;
                        }

        private:
                Layout <MaxAnalogPoles> m_analogStorage = {};
                Layout <MaxDigitalPoles> m_digitalStorage = {};
        };

//------------------------------------------------------------------------------

        class IIR_EXPORT LowPassTransform
        {
        public:
        LowPassTransform (double fc,
                          LayoutBase& digital,
                          LayoutBase const& analog);

        private:
        complex_t transform (complex_t c);

        double f = 0.0;
        };

//------------------------------------------------------------------------------

        class IIR_EXPORT HighPassTransform
        {
        public:
        HighPassTransform (double fc,
                           LayoutBase& digital,
                           LayoutBase const& analog);

        private:
        complex_t transform (complex_t c);

        double f = 0.0;
        };

//------------------------------------------------------------------------------

        class IIR_EXPORT BandPassTransform
        {

        public:
        BandPassTransform (double fc,
                           double fw,
                           LayoutBase& digital,
                           LayoutBase const& analog);

        private:
        ComplexPair transform (complex_t c);

        double wc = 0.0;
        double wc2 = 0.0;
        double a = 0.0;
        double b = 0.0;
        double a2 = 0.0;
        double b2 = 0.0;
        double ab = 0.0;
        double ab_2 = 0.0;
        };

//------------------------------------------------------------------------------

        class IIR_EXPORT BandStopTransform
        {
        public:
        BandStopTransform (double fc,
                           double fw,
                           LayoutBase& digital,
                           LayoutBase const& analog);

        private:
        ComplexPair transform (complex_t c);

        double wc = 0.0;
        double wc2 = 0.0;
        double a = 0.0;
        double b = 0.0;
        double a2 = 0.0;
        double b2 = 0.0;
        };

}

#endif
```

8.12 RBJ.h

```

#ifndef IIR1_RBJ_H
#define IIR1_RBJ_H

#include "Common.h"
#include "Biquad.h"
#include "State.h"

namespace Iir {

#define ONESQRT2 (1/sqrt(2))

namespace RBJ {

        struct IIR_EXPORT RBJbase : Biquad
        {
        public:
                template <typename Sample>
                        inline Sample filter(Sample s) {
                        return static_cast<Sample>(state.filter((double)s,*this));
                }
                void reset() {
                        state.reset();
                }
                const DirectFormI& getState() {
                        return state;
                }
        private:
                DirectFormI state = {};
        };

        struct IIR_EXPORT LowPass : RBJbase
        {
                void setupN(double cutoffFrequency,
                            double q = ONESQRT2);

                void setup(double sampleRate,
                           double cutoffFrequency,
                           double q = ONESQRT2) {
                        setupN(cutoffFrequency / sampleRate, q);
                }
        };

        struct IIR_EXPORT HighPass : RBJbase
        {
                void setupN(double cutoffFrequency,
                            double q = ONESQRT2);
                void setup (double sampleRate,
                            double cutoffFrequency,
                            double q = ONESQRT2) {
                        setupN(cutoffFrequency / sampleRate, q);
                }
        };

        struct IIR_EXPORT BandPass1 : RBJbase
        {
                void setupN(double centerFrequency,
                            double bandWidth);
                void setup (double sampleRate,
                            double centerFrequency,
                            double bandWidth) {
                        setupN(centerFrequency / sampleRate, bandWidth);
                }
        };

        struct IIR_EXPORT BandPass2 : RBJbase
        {
                void setupN(double centerFrequency,
                            double bandWidth);
                void setup (double sampleRate,
                            double centerFrequency,
                            double bandWidth) {
                        setupN(centerFrequency / sampleRate, bandWidth);
                }
        };

        struct IIR_EXPORT BandStop : RBJbase
        {
                void setupN(double centerFrequency,
                            double bandWidth);
                void setup (double sampleRate,
                            double centerFrequency,
                            double bandWidth) {
                        setupN(centerFrequency / sampleRate, bandWidth);
                }
        };

        struct IIR_EXPORT IIRNotch : RBJbase
        {
                void setupN(double centerFrequency,
                            double q_factor = 10);
                void setup (double sampleRate,
                            double centerFrequency,
                            double q_factor = 10) {
                        setupN(centerFrequency / sampleRate, q_factor);
                }
        };

        struct IIR_EXPORT LowShelf : RBJbase
        {
                void setupN(double cutoffFrequency,
                            double gainDb,
                            double shelfSlope = 1);
                void setup (double sampleRate,
                            double cutoffFrequency,
                            double gainDb,
                            double shelfSlope = 1) {
                        setupN( cutoffFrequency / sampleRate, gainDb, shelfSlope);
                }
        };

        struct IIR_EXPORT HighShelf : RBJbase
        {
                void setupN(double cutoffFrequency,
                            double gainDb,
                            double shelfSlope = 1);
                void setup (double sampleRate,
                            double cutoffFrequency,
                            double gainDb,
                            double shelfSlope = 1) {
                        setupN( cutoffFrequency / sampleRate, gainDb, shelfSlope);
                }
        };

        struct IIR_EXPORT BandShelf : RBJbase
        {
                void setupN(double centerFrequency,
                            double gainDb,
                            double bandWidth);
                void setup (double sampleRate,
                            double centerFrequency,
                            double gainDb,
                            double bandWidth) {
                        setupN(centerFrequency / sampleRate, gainDb, bandWidth);
                }
        };

        struct IIR_EXPORT AllPass : RBJbase
        {
                void setupN(double phaseFrequency,
                            double q  = ONESQRT2);

                void setup (double sampleRate,
                            double phaseFrequency,
                            double q  = ONESQRT2) {
                        setupN( phaseFrequency / sampleRate, q);
                }
        };

}

}


#endif
```

8.13 State.h

```
#ifndef IIR1_STATE_H
#define IIR1_STATE_H

#include "Common.h"
#include "Biquad.h"


#define DEFAULT_STATE DirectFormII

namespace Iir {

        class IIR_EXPORT DirectFormI
        {
        public:
        DirectFormI () = default;

        void reset ()
        {
                m_x1 = 0;
                m_x2 = 0;
                m_y1 = 0;
                m_y2 = 0;
        }

        inline double filter(const double in,
                             const Biquad& s)
        {
                const double out = s.m_b0*in + s.m_b1*m_x1 + s.m_b2*m_x2
                - s.m_a1*m_y1 - s.m_a2*m_y2;
                m_x2 = m_x1;
                m_y2 = m_y1;
                m_x1 = in;
                m_y1 = out;

                return out;
        }

        protected:
        double m_x2 = 0.0; // x[n-2]
        double m_y2 = 0.0; // y[n-2]
        double m_x1 = 0.0; // x[n-1]
        double m_y1 = 0.0; // y[n-1]
        };

//------------------------------------------------------------------------------

        class IIR_EXPORT DirectFormII
        {
        public:
        DirectFormII () = default;

        void reset ()
        {
                m_v1 = 0.0;
                m_v2 = 0.0;
        }

        inline double filter(const double in,
                             const Biquad& s)
        {
                const double w   = in - s.m_a1*m_v1 - s.m_a2*m_v2;
                const double out =      s.m_b0*w    + s.m_b1*m_v1 + s.m_b2*m_v2;

                m_v2 = m_v1;
                m_v1 = w;

                return out;
        }

        private:
        double m_v1 = 0.0; // v[-1]
        double m_v2 = 0.0; // v[-2]
        };


//------------------------------------------------------------------------------

        class IIR_EXPORT TransposedDirectFormII
        {
        public:
        TransposedDirectFormII() = default;

        void reset ()
        {
                m_s1 = 0.0;
                m_s1_1 = 0.0;
                m_s2 = 0.0;
                m_s2_1 = 0.0;
        }

        inline double filter(const double in,
                             const Biquad& s)
        {
                const double out = m_s1_1 + s.m_b0*in;
                m_s1 = m_s2_1 + s.m_b1*in - s.m_a1*out;
                m_s2 = s.m_b2*in - s.m_a2*out;
                m_s1_1 = m_s1;
                m_s2_1 = m_s2;

                return out;
        }

        private:
        double m_s1 = 0.0;
        double m_s1_1 = 0.0;
        double m_s2 = 0.0;
        double m_s2_1 = 0.0;
        };

}

#endif
```

8.14 Types.h

```

#ifndef IIR1_TYPES_H
#define IIR1_TYPES_H

#include "Common.h"
#include "MathSupplement.h"

namespace Iir {

        struct IIR_EXPORT ComplexPair : complex_pair_t
        {
                ComplexPair() = default;

                explicit ComplexPair (const complex_t& c1)
                        : complex_pair_t (c1, 0.)
                {
                        if (!isReal()) throw_invalid_argument("A single complex number needs to be
real.");
                }

                ComplexPair (const complex_t& c1,
                             const complex_t& c2)
                        : complex_pair_t (c1, c2)
                {
                }

                bool isReal () const
                {
                        return first.imag() == 0 && second.imag() == 0;
                }

                bool isMatchedPair () const
                {
                        if (first.imag() != 0)
                                return second == std::conj (first);
                        else
                                return second.imag () == 0 &&
                                        second.real () != 0 &&
                                        first.real () != 0;
                }

                bool is_nan () const
                {
                        return Iir::is_nan (first) || Iir::is_nan (second);
                }
        };


        struct IIR_EXPORT PoleZeroPair
        {
                ComplexPair poles = ComplexPair();
                ComplexPair zeros = ComplexPair();

                PoleZeroPair () = default;

```

```
            // single pole/zero
            PoleZeroPair (const complex_t& p, const complex_t& z)
                    : poles (p), zeros (z)
            {
            }

            // pole/zero pair
            PoleZeroPair (const complex_t& p1, const complex_t& z1,
                          const complex_t& p2, const complex_t& z2)
                    : poles (p1, p2)
                    , zeros (z1, z2)
            {
            }

            bool isSinglePole () const
            {
                    return poles.second == 0. && zeros.second == 0.;
            }

            bool is_nan () const
            {
                    return poles.is_nan() || zeros.is_nan();
            }
    };


}

#endif
```

Index