

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 | 1 |
| 2 Namespace Index | 1 |
| 2.1 Namespace List | 1 |
| 3 Hierarchical Index | 1 |
| 3.1 Class Hierarchy | 1 |
| 4 Class Index | 5 |
| 4.1 Class List | 5 |
| 5 File Index | 8 |
| 5.1 File List | 8 |
| 6 Namespace Documentation | 8 |
| 6.1 Iir Namespace Reference | 8 |
| 6.1.1 Detailed Description | 9 |
| 6.2 Iir::Butterworth Namespace Reference | 11 |
| 6.2.1 Detailed Description | 11 |
| 6.3 Iir::ChebyshevI Namespace Reference | 11 |
| 6.3.1 Detailed Description | 11 |
| 6.4 Iir::ChebyshevII Namespace Reference | 12 |
| 6.4.1 Detailed Description | 12 |
| 6.5 Iir::Custom Namespace Reference | 12 |
| 6.5.1 Detailed Description | 12 |
| 7 Class Documentation | 12 |
| 7.1 Iir::RBJ::AllPass Struct Reference | 12 |
| 7.1.1 Detailed Description | 13 |
| 7.1.2 Member Function Documentation | 13 |
| 7.2 Iir::Butterworth::AnalogLowPass Class Reference | 14 |
| 7.2.1 Detailed Description | 14 |
| 7.3 Iir::ChebyshevI::AnalogLowPass Class Reference | 14 |
| 7.3.1 Detailed Description | 14 |
| 7.4 Iir::ChebyshevII::AnalogLowPass Class Reference | 14 |
| 7.4.1 Detailed Description | 15 |
| 7.5 Iir::Butterworth::AnalogLowShelf Class Reference | 15 |
| 7.5.1 Detailed Description | 15 |
| 7.6 Iir::ChebyshevI::AnalogLowShelf Class Reference | 15 |
| 7.6.1 Detailed Description | 15 |
| 7.7 Iir::ChebyshevII::AnalogLowShelf Class Reference | 15 |
| 7.7.1 Detailed Description | 16 |
| 7.8 Iir::Butterworth::BandPass< FilterOrder, StateType > Struct Template Reference | 16 |
| 7.8.1 Detailed Description | 16 |

| | |
|--|----|
| 7.8.2 Member Function Documentation | 16 |
| 7.9 lir::ChebyshevI::BandPass< FilterOrder, StateType > Struct Template Reference | 18 |
| 7.9.1 Detailed Description | 18 |
| 7.9.2 Member Function Documentation | 18 |
| 7.10 lir::ChebyshevII::BandPass< FilterOrder, StateType > Struct Template Reference | 20 |
| 7.10.1 Detailed Description | 20 |
| 7.10.2 Member Function Documentation | 20 |
| 7.11 lir::RBJ::BandPass1 Struct Reference | 22 |
| 7.11.1 Detailed Description | 23 |
| 7.11.2 Member Function Documentation | 23 |
| 7.12 lir::RBJ::BandPass2 Struct Reference | 23 |
| 7.12.1 Detailed Description | 24 |
| 7.12.2 Member Function Documentation | 24 |
| 7.13 lir::Butterworth::BandPassBase Struct Reference | 25 |
| 7.14 lir::ChebyshevI::BandPassBase Struct Reference | 25 |
| 7.15 lir::ChebyshevII::BandPassBase Struct Reference | 26 |
| 7.16 lir::BandPassTransform Class Reference | 26 |
| 7.16.1 Detailed Description | 26 |
| 7.17 lir::Butterworth::BandShelf< FilterOrder, StateType > Struct Template Reference | 26 |
| 7.17.1 Detailed Description | 27 |
| 7.17.2 Member Function Documentation | 27 |
| 7.18 lir::ChebyshevI::BandShelf< FilterOrder, StateType > Struct Template Reference | 28 |
| 7.18.1 Detailed Description | 29 |
| 7.18.2 Member Function Documentation | 29 |
| 7.19 lir::ChebyshevII::BandShelf< FilterOrder, StateType > Struct Template Reference | 31 |
| 7.19.1 Detailed Description | 31 |
| 7.19.2 Member Function Documentation | 31 |
| 7.20 lir::RBJ::BandShelf Struct Reference | 33 |
| 7.20.1 Detailed Description | 34 |
| 7.20.2 Member Function Documentation | 34 |
| 7.21 lir::Butterworth::BandShelfBase Struct Reference | 35 |
| 7.22 lir::ChebyshevI::BandShelfBase Struct Reference | 35 |
| 7.23 lir::ChebyshevII::BandShelfBase Struct Reference | 36 |
| 7.24 lir::Butterworth::BandStop< FilterOrder, StateType > Struct Template Reference | 36 |
| 7.24.1 Detailed Description | 37 |
| 7.24.2 Member Function Documentation | 37 |
| 7.25 lir::ChebyshevI::BandStop< FilterOrder, StateType > Struct Template Reference | 38 |
| 7.25.1 Detailed Description | 39 |
| 7.25.2 Member Function Documentation | 39 |
| 7.26 lir::ChebyshevII::BandStop< FilterOrder, StateType > Struct Template Reference | 40 |
| 7.26.1 Detailed Description | 41 |
| 7.26.2 Member Function Documentation | 41 |

| | |
|---|----|
| 7.27 lir::RBJ::BandStop Struct Reference | 42 |
| 7.27.1 Detailed Description | 43 |
| 7.27.2 Member Function Documentation | 43 |
| 7.28 lir::Butterworth::BandStopBase Struct Reference | 44 |
| 7.29 lir::ChebyshevI::BandStopBase Struct Reference | 44 |
| 7.30 lir::ChebyshevII::BandStopBase Struct Reference | 45 |
| 7.31 lir::BandStopTransform Class Reference | 45 |
| 7.31.1 Detailed Description | 45 |
| 7.32 lir::Biquad Class Reference | 45 |
| 7.32.1 Member Function Documentation | 46 |
| 7.33 lir::BiquadPoleState Struct Reference | 49 |
| 7.33.1 Detailed Description | 49 |
| 7.34 lir::Cascade Class Reference | 49 |
| 7.34.1 Detailed Description | 50 |
| 7.34.2 Member Function Documentation | 50 |
| 7.35 lir::CascadeStages< MaxStages, StateType > Class Template Reference | 50 |
| 7.35.1 Detailed Description | 51 |
| 7.35.2 Member Function Documentation | 51 |
| 7.36 lir::ComplexPair Struct Reference | 52 |
| 7.36.1 Detailed Description | 52 |
| 7.36.2 Member Function Documentation | 52 |
| 7.37 lir::DirectFormI Class Reference | 52 |
| 7.37.1 Detailed Description | 52 |
| 7.38 lir::DirectFormII Class Reference | 53 |
| 7.38.1 Detailed Description | 53 |
| 7.39 lir::Butterworth::HighPass< FilterOrder, StateType > Struct Template Reference | 53 |
| 7.39.1 Detailed Description | 53 |
| 7.39.2 Member Function Documentation | 54 |
| 7.40 lir::ChebyshevI::HighPass< FilterOrder, StateType > Struct Template Reference | 55 |
| 7.40.1 Detailed Description | 55 |
| 7.40.2 Member Function Documentation | 55 |
| 7.41 lir::ChebyshevII::HighPass< FilterOrder, StateType > Struct Template Reference | 56 |
| 7.41.1 Detailed Description | 57 |
| 7.41.2 Member Function Documentation | 57 |
| 7.42 lir::RBJ::HighPass Struct Reference | 58 |
| 7.42.1 Detailed Description | 59 |
| 7.42.2 Member Function Documentation | 59 |
| 7.43 lir::Butterworth::HighPassBase Struct Reference | 60 |
| 7.44 lir::ChebyshevI::HighPassBase Struct Reference | 60 |
| 7.45 lir::ChebyshevII::HighPassBase Struct Reference | 61 |
| 7.46 lir::HighPassTransform Class Reference | 61 |
| 7.46.1 Detailed Description | 62 |

| | |
|--|----|
| 7.47 iir::Butterworth::HighShelf< FilterOrder, StateType > Struct Template Reference | 62 |
| 7.47.1 Detailed Description | 62 |
| 7.47.2 Member Function Documentation | 62 |
| 7.48 iir::ChebyshevI::HighShelf< FilterOrder, StateType > Struct Template Reference | 64 |
| 7.48.1 Detailed Description | 64 |
| 7.48.2 Member Function Documentation | 64 |
| 7.49 iir::ChebyshevII::HighShelf< FilterOrder, StateType > Struct Template Reference | 66 |
| 7.49.1 Detailed Description | 66 |
| 7.49.2 Member Function Documentation | 66 |
| 7.50 iir::RBJ::HighShelf Struct Reference | 68 |
| 7.50.1 Detailed Description | 68 |
| 7.50.2 Member Function Documentation | 69 |
| 7.51 iir::Butterworth::HighShelfBase Struct Reference | 69 |
| 7.52 iir::ChebyshevI::HighShelfBase Struct Reference | 70 |
| 7.53 iir::ChebyshevII::HighShelfBase Struct Reference | 70 |
| 7.54 iir::RBJ::IIRNotch Struct Reference | 71 |
| 7.54.1 Detailed Description | 72 |
| 7.54.2 Member Function Documentation | 72 |
| 7.55 iir::Layout< MaxPoles > Class Template Reference | 72 |
| 7.55.1 Detailed Description | 72 |
| 7.56 iir::LayoutBase Class Reference | 72 |
| 7.56.1 Detailed Description | 73 |
| 7.57 iir::Butterworth::LowPass< FilterOrder, StateType > Struct Template Reference | 73 |
| 7.57.1 Detailed Description | 73 |
| 7.57.2 Member Function Documentation | 73 |
| 7.58 iir::ChebyshevI::LowPass< FilterOrder, StateType > Struct Template Reference | 74 |
| 7.58.1 Detailed Description | 75 |
| 7.58.2 Member Function Documentation | 75 |
| 7.59 iir::ChebyshevII::LowPass< FilterOrder, StateType > Struct Template Reference | 76 |
| 7.59.1 Detailed Description | 77 |
| 7.59.2 Member Function Documentation | 77 |
| 7.60 iir::RBJ::LowPass Struct Reference | 78 |
| 7.60.1 Detailed Description | 79 |
| 7.60.2 Member Function Documentation | 79 |
| 7.61 iir::Butterworth::LowPassBase Struct Reference | 80 |
| 7.62 iir::ChebyshevI::LowPassBase Struct Reference | 80 |
| 7.63 iir::ChebyshevII::LowPassBase Struct Reference | 81 |
| 7.64 iir::LowPassTransform Class Reference | 81 |
| 7.64.1 Detailed Description | 82 |
| 7.65 iir::Butterworth::LowShelf< FilterOrder, StateType > Struct Template Reference | 82 |
| 7.65.1 Detailed Description | 82 |
| 7.65.2 Member Function Documentation | 82 |

| | |
|---|-----------|
| 7.66 lir::ChebyshevI::LowShelf< FilterOrder, StateType > Struct Template Reference | 84 |
| 7.66.1 Detailed Description | 84 |
| 7.66.2 Member Function Documentation | 84 |
| 7.67 lir::ChebyshevII::LowShelf< FilterOrder, StateType > Struct Template Reference | 86 |
| 7.67.1 Detailed Description | 86 |
| 7.67.2 Member Function Documentation | 86 |
| 7.68 lir::RBJ::LowShelf Struct Reference | 88 |
| 7.68.1 Detailed Description | 89 |
| 7.68.2 Member Function Documentation | 89 |
| 7.69 lir::Butterworth::LowShelfBase Struct Reference | 89 |
| 7.70 lir::ChebyshevI::LowShelfBase Struct Reference | 90 |
| 7.71 lir::ChebyshevII::LowShelfBase Struct Reference | 90 |
| 7.72 lir::Custom::OnePole Struct Reference | 91 |
| 7.72.1 Detailed Description | 92 |
| 7.73 lir::PoleFilter< BaseClass, StateType, MaxAnalogPoles, MaxDigitalPoles > Struct Template Reference | 93 |
| 7.73.1 Detailed Description | 93 |
| 7.74 lir::PoleFilterBase< AnalogPrototype > Class Template Reference | 94 |
| 7.74.1 Detailed Description | 94 |
| 7.75 lir::PoleFilterBase2 Class Reference | 94 |
| 7.75.1 Detailed Description | 95 |
| 7.76 lir::PoleZeroPair Struct Reference | 95 |
| 7.76.1 Detailed Description | 95 |
| 7.77 lir::RBJ::RBJbase Struct Reference | 96 |
| 7.77.1 Detailed Description | 97 |
| 7.78 lir::Custom::SOSCascade< NSOS, StateType > Struct Template Reference | 97 |
| 7.78.1 Detailed Description | 97 |
| 7.78.2 Constructor & Destructor Documentation | 97 |
| 7.78.3 Member Function Documentation | 98 |
| 7.79 lir::Cascade::Storage Struct Reference | 98 |
| 7.79.1 Detailed Description | 98 |
| 7.80 lir::TransposedDirectFormII Class Reference | 98 |
| 7.81 lir::Custom::TwoPole Struct Reference | 98 |
| 7.81.1 Detailed Description | 99 |
| 8 File Documentation | 99 |
| 8.1 lir.h | 99 |
| 8.2 Biquad.h | 100 |
| 8.3 Butterworth.h | 101 |
| 8.4 Cascade.h | 105 |
| 8.5 ChebyshevI.h | 106 |
| 8.6 ChebyshevII.h | 112 |
| 8.7 Common.h | 117 |

| | |
|---------------------------------|------------|
| 8.8 Custom.h | 118 |
| 8.9 Layout.h | 118 |
| 8.10 MathSupplement.h | 120 |
| 8.11 PoleFilter.h | 121 |
| 8.12 RBJ.h | 123 |
| 8.13 State.h | 125 |
| 8.14 Types.h | 126 |
| Index | 129 |

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:

| | |
|-------------------------|-----------|
| lir | 8 |
| lir::Butterworth | 11 |
| lir::ChebyshevI | 11 |
| lir::ChebyshevII | 12 |
| lir::Custom | 12 |

3 Hierarchical Index

3.1 Class Hierarchy

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

BandPassBase

| | |
|---|-----------|
| lir::PoleFilter< BandPassBase, DirectFormII, 4, 4 *2 > | 93 |
| lir::BandPassTransform | 26 |
| BandShelfBase | |
| lir::PoleFilter< BandShelfBase, DirectFormII, 4, 4 *2 > | 93 |
| BandStopBase | |
| lir::PoleFilter< BandStopBase, DirectFormII, 4, 4 *2 > | 93 |
| lir::BandStopTransform | 45 |
| BaseClass | |
| lir::PoleFilter< BaseClass, StateType, MaxAnalogPoles, MaxDigitalPoles > | 93 |
| lir::Butterworth::BandPass< FilterOrder, StateType > | 16 |
| lir::Butterworth::BandShelf< FilterOrder, StateType > | 26 |
| lir::Butterworth::BandStop< FilterOrder, StateType > | 36 |
| lir::Butterworth::HighPass< FilterOrder, StateType > | 53 |
| lir::Butterworth::HighShelf< FilterOrder, StateType > | 62 |
| lir::Butterworth::LowPass< FilterOrder, StateType > | 73 |
| lir::Butterworth::LowShelf< FilterOrder, StateType > | 82 |
| lir::ChebyshevI::BandPass< FilterOrder, StateType > | 18 |
| lir::ChebyshevI::BandShelf< FilterOrder, StateType > | 28 |
| lir::ChebyshevI::BandStop< FilterOrder, StateType > | 38 |
| lir::ChebyshevI::HighPass< FilterOrder, StateType > | 55 |
| lir::ChebyshevI::HighShelf< FilterOrder, StateType > | 64 |
| lir::ChebyshevI::LowPass< FilterOrder, StateType > | 74 |
| lir::ChebyshevI::LowShelf< FilterOrder, StateType > | 84 |
| lir::ChebyshevII::BandPass< FilterOrder, StateType > | 20 |
| lir::ChebyshevII::BandShelf< FilterOrder, StateType > | 31 |
| lir::ChebyshevII::BandStop< FilterOrder, StateType > | 40 |
| lir::ChebyshevII::HighPass< FilterOrder, StateType > | 56 |
| lir::ChebyshevII::HighShelf< FilterOrder, StateType > | 66 |
| lir::ChebyshevII::LowPass< FilterOrder, StateType > | 76 |
| lir::ChebyshevII::LowShelf< FilterOrder, StateType > | 86 |
| lir::Biquad | 45 |
| lir::Custom::OnePole | 91 |
| lir::Custom::TwoPole | 98 |

| | |
|---------------------------------------|----|
| lir::RBJ::RBJbase | 96 |
| lir::RBJ::AllPass | 12 |
| lir::RBJ::BandPass1 | 22 |
| lir::RBJ::BandPass2 | 23 |
| lir::RBJ::BandShelf | 33 |
| lir::RBJ::BandStop | 42 |
| lir::RBJ::HighPass | 58 |
| lir::RBJ::HighShelf | 68 |
| lir::RBJ::IIRNotch | 71 |
| lir::RBJ::LowPass | 78 |
| lir::RBJ::LowShelf | 88 |
| lir::Cascade | 49 |
| lir::PoleFilterBase2 | 94 |
| lir::PoleFilterBase< AnalogLowPass > | 94 |
| lir::Butterworth::BandPassBase | 25 |
| lir::Butterworth::BandStopBase | 44 |
| lir::Butterworth::HighPassBase | 60 |
| lir::Butterworth::LowPassBase | 80 |
| lir::ChebyshevI::BandPassBase | 25 |
| lir::ChebyshevI::BandStopBase | 44 |
| lir::ChebyshevI::HighPassBase | 60 |
| lir::ChebyshevI::LowPassBase | 80 |
| lir::ChebyshevII::BandPassBase | 26 |
| lir::ChebyshevII::BandStopBase | 45 |
| lir::ChebyshevII::HighPassBase | 61 |
| lir::ChebyshevII::LowPassBase | 81 |
| lir::PoleFilterBase< AnalogLowShelf > | 94 |
| lir::Butterworth::BandShelfBase | 35 |
| lir::Butterworth::HighShelfBase | 69 |
| lir::Butterworth::LowShelfBase | 89 |
| lir::ChebyshevI::BandShelfBase | 35 |
| lir::ChebyshevI::HighShelfBase | 70 |

| | |
|--|----|
| lir::ChebyshevI::LowShelfBase | 90 |
| lir::ChebyshevII::BandShelfBase | 36 |
| lir::ChebyshevII::HighShelfBase | 70 |
| lir::ChebyshevII::LowShelfBase | 90 |
| lir::PoleFilterBase< AnalogPrototype > | 94 |
| lir::CascadeStages< MaxStages, StateType > | 50 |
| lir::PoleFilter< BandPassBase, DirectFormII, 4, 4 *2 > | 93 |
| lir::PoleFilter< BandShelfBase, DirectFormII, 4, 4 *2 > | 93 |
| lir::PoleFilter< BandStopBase, DirectFormII, 4, 4 *2 > | 93 |
| lir::PoleFilter< HighPassBase, DirectFormII, 4 > | 93 |
| lir::PoleFilter< HighShelfBase, DirectFormII, 4 > | 93 |
| lir::PoleFilter< LowPassBase, DirectFormII, 4 > | 93 |
| lir::PoleFilter< LowShelfBase, DirectFormII, 4 > | 93 |
| lir::Custom::SOSCascade< NSOS, StateType > | 97 |
| lir::PoleFilter< BaseClass, StateType, MaxAnalogPoles, MaxDigitalPoles > | 93 |
| lir::CascadeStages< NSOS, DirectFormII > | 50 |
| lir::CascadeStages<(MaxAnalogPoles+1)/2, DirectFormII > | 50 |
| lir::CascadeStages<(MaxAnalogPoles+1)/2, StateType > | 50 |
| lir::CascadeStages<(MaxDigitalPoles+1)/2, DirectFormII > | 50 |
| complex_pair_t | |
| lir::ComplexPair | 52 |
| lir::DirectFormI | 52 |
| lir::DirectFormII | 53 |
| HighPassBase | |
| lir::PoleFilter< HighPassBase, DirectFormII, 4 > | 93 |
| lir::HighPassTransform | 61 |
| HighShelfBase | |
| lir::PoleFilter< HighShelfBase, DirectFormII, 4 > | 93 |
| lir::Layout< MaxPoles > | 72 |
| lir::Layout< MaxAnalogPoles > | 72 |
| lir::Layout< MaxDigitalPoles > | 72 |
| lir::LayoutBase | 72 |
| lir::Butterworth::AnalogLowPass | 14 |

| | |
|--|----|
| lir::Butterworth::AnalogLowShelf | 15 |
| lir::ChebyshevI::AnalogLowPass | 14 |
| lir::ChebyshevI::AnalogLowShelf | 15 |
| lir::ChebyshevII::AnalogLowPass | 14 |
| lir::ChebyshevII::AnalogLowShelf | 15 |
| LowPassBase | |
| lir::PoleFilter< LowPassBase, DirectFormII, 4 > | 93 |
| lir::LowPassTransform | 81 |
| LowShelfBase | |
| lir::PoleFilter< LowShelfBase, DirectFormII, 4 > | 93 |
| lir::PoleZeroPair | 95 |
| lir::BiquadPoleState | 49 |
| lir::Cascade::Storage | 98 |
| lir::TransposedDirectFormII | 98 |

4 Class Index

4.1 Class List

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

| | |
|--|----|
| lir::RBJ::AllPass | 12 |
| lir::Butterworth::AnalogLowPass | 14 |
| lir::ChebyshevI::AnalogLowPass | 14 |
| lir::ChebyshevII::AnalogLowPass | 14 |
| lir::Butterworth::AnalogLowShelf | 15 |
| lir::ChebyshevI::AnalogLowShelf | 15 |
| lir::ChebyshevII::AnalogLowShelf | 15 |
| lir::Butterworth::BandPass< FilterOrder, StateType > | 16 |
| lir::ChebyshevI::BandPass< FilterOrder, StateType > | 18 |
| lir::ChebyshevII::BandPass< FilterOrder, StateType > | 20 |
| lir::RBJ::BandPass1 | 22 |
| lir::RBJ::BandPass2 | 23 |
| lir::Butterworth::BandPassBase | 25 |
| lir::ChebyshevI::BandPassBase | 25 |

| | |
|--|----|
| <code>lir::ChebyshevII::BandPassBase</code> | 26 |
| <code>lir::BandPassTransform</code> | 26 |
| <code>lir::Butterworth::BandShelf< FilterOrder, StateType ></code> | 26 |
| <code>lir::ChebyshevI::BandShelf< FilterOrder, StateType ></code> | 28 |
| <code>lir::ChebyshevII::BandShelf< FilterOrder, StateType ></code> | 31 |
| <code>lir::RBJ::BandShelf</code> | 33 |
| <code>lir::Butterworth::BandShelfBase</code> | 35 |
| <code>lir::ChebyshevI::BandShelfBase</code> | 35 |
| <code>lir::ChebyshevII::BandShelfBase</code> | 36 |
| <code>lir::Butterworth::BandStop< FilterOrder, StateType ></code> | 36 |
| <code>lir::ChebyshevI::BandStop< FilterOrder, StateType ></code> | 38 |
| <code>lir::ChebyshevII::BandStop< FilterOrder, StateType ></code> | 40 |
| <code>lir::RBJ::BandStop</code> | 42 |
| <code>lir::Butterworth::BandStopBase</code> | 44 |
| <code>lir::ChebyshevI::BandStopBase</code> | 44 |
| <code>lir::ChebyshevII::BandStopBase</code> | 45 |
| <code>lir::BandStopTransform</code> | 45 |
| <code>lir::Biquad</code> | 45 |
| <code>lir::BiquadPoleState</code> | 49 |
| <code>lir::Cascade</code> | 49 |
| <code>lir::CascadeStages< MaxStages, StateType ></code> | 50 |
| <code>lir::ComplexPair</code> | 52 |
| <code>lir::DirectFormI</code> | 52 |
| <code>lir::DirectFormII</code> | 53 |
| <code>lir::Butterworth::HighPass< FilterOrder, StateType ></code> | 53 |
| <code>lir::ChebyshevI::HighPass< FilterOrder, StateType ></code> | 55 |
| <code>lir::ChebyshevII::HighPass< FilterOrder, StateType ></code> | 56 |
| <code>lir::RBJ::HighPass</code> | 58 |
| <code>lir::Butterworth::HighPassBase</code> | 60 |
| <code>lir::ChebyshevI::HighPassBase</code> | 60 |
| <code>lir::ChebyshevII::HighPassBase</code> | 61 |
| <code>lir::HighPassTransform</code> | 61 |

| | |
|--|----|
| lir::Butterworth::HighShelf< FilterOrder, StateType > | 62 |
| lir::ChebyshevI::HighShelf< FilterOrder, StateType > | 64 |
| lir::ChebyshevII::HighShelf< FilterOrder, StateType > | 66 |
| lir::RBJ::HighShelf | 68 |
| lir::Butterworth::HighShelfBase | 69 |
| lir::ChebyshevI::HighShelfBase | 70 |
| lir::ChebyshevII::HighShelfBase | 70 |
| lir::RBJ::IIRNotch | 71 |
| lir::Layout< MaxPoles > | 72 |
| lir::LayoutBase | 72 |
| lir::Butterworth::LowPass< FilterOrder, StateType > | 73 |
| lir::ChebyshevI::LowPass< FilterOrder, StateType > | 74 |
| lir::ChebyshevII::LowPass< FilterOrder, StateType > | 76 |
| lir::RBJ::LowPass | 78 |
| lir::Butterworth::LowPassBase | 80 |
| lir::ChebyshevI::LowPassBase | 80 |
| lir::ChebyshevII::LowPassBase | 81 |
| lir::LowPassTransform | 81 |
| lir::Butterworth::LowShelf< FilterOrder, StateType > | 82 |
| lir::ChebyshevI::LowShelf< FilterOrder, StateType > | 84 |
| lir::ChebyshevII::LowShelf< FilterOrder, StateType > | 86 |
| lir::RBJ::LowShelf | 88 |
| lir::Butterworth::LowShelfBase | 89 |
| lir::ChebyshevI::LowShelfBase | 90 |
| lir::ChebyshevII::LowShelfBase | 90 |
| lir::Custom::OnePole | 91 |
| lir::PoleFilter< BaseClass, StateType, MaxAnalogPoles, MaxDigitalPoles > | 93 |
| lir::PoleFilterBase< AnalogPrototype > | 94 |
| lir::PoleFilterBase2 | 94 |
| lir::PoleZeroPair | 95 |
| lir::RBJ::RBJbase | 96 |
| lir::Custom::SOSCascade< NSOS, StateType > | 97 |

| | |
|---|----|
| lir::Cascade::Storage | 98 |
| lir::TransposedDirectFormII | 98 |
| lir::Custom::TwoPole | 98 |

5 File Index

5.1 File List

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

| | |
|--------------------------------------|-----|
| lir.h | 99 |
| iir/Biquad.h | 100 |
| iir/Butterworth.h | 101 |
| iir/Cascade.h | 105 |
| iir/ChebyshevI.h | 106 |
| iir/ChebyshevII.h | 112 |
| iir/Common.h | 117 |
| iir/Custom.h | 118 |
| iir/Layout.h | 118 |
| iir/MathSupplement.h | 120 |
| iir/PoleFilter.h | 121 |
| iir/RBJ.h | 123 |
| iir/State.h | 125 |
| iir/Types.h | 126 |

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.

License: MIT License (<http://www.opensource.org/licenses/mit-license.php>) Copyright (c) 2009 by Vinnie Falco Copyright (c) 2011 by Bernd Porr

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

"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.

License: MIT License (<http://www.opensource.org/licenses/mit-license.php>) Copyright (c) 2009 by Vinnie Falco Copyright (c) 2011-2021 by Bernd Porr

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE

LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

"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.

License: MIT License (<http://www.opensource.org/licenses/mit-license.php>) Copyright (c) 2009 by Vinnie Falco Copyright (c) 2011-2021 by Bernd Porr

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

"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.

License: MIT License (<http://www.opensource.org/licenses/mit-license.php>) Copyright (c) 2009 by Vinnie Falco Copyright (c) 2011 by Bernd Porr

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

"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.

License: MIT License (<http://www.opensource.org/licenses/mit-license.php>) Copyright (c) 2009 by Vinnie Falco Copyright (c) 2011 by Bernd Porr

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. 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 `lir::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 provide the filter order at setup time to force a lower order than the default one.

6.3 `lir::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

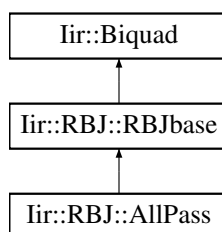
Single pole, [Biquad](#) and cascade of Biquads with parameters allowing for directly setting the parameters.

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

| | |
|-----------------------|---------------------------------|
| <i>sampleRate</i> | Sampling rate |
| <i>phaseFrequency</i> | Frequency where the phase flips |
| <i>q</i> | Q-factor |

setupN()

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

Calculates the coefficients

Parameters

| | |
|-----------------------|--|
| <i>phaseFrequency</i> | Normalised frequency where the phase flips |
| <i>q</i> | Q-factor |

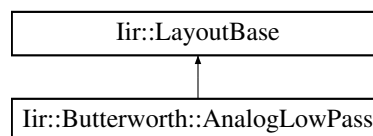
The documentation for this struct was generated from the following files:

- 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)

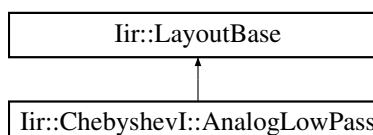
The documentation for this class was generated from the following files:

- 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)

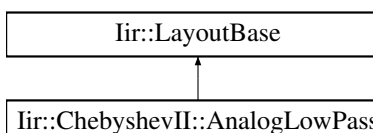
The documentation for this class was generated from the following files:

- 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)

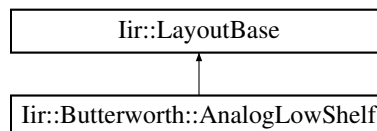
The documentation for this class was generated from the following files:

- 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)

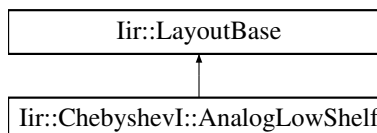
The documentation for this class was generated from the following files:

- 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)

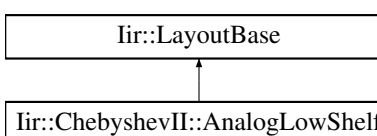
The documentation for this class was generated from the following files:

- 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)

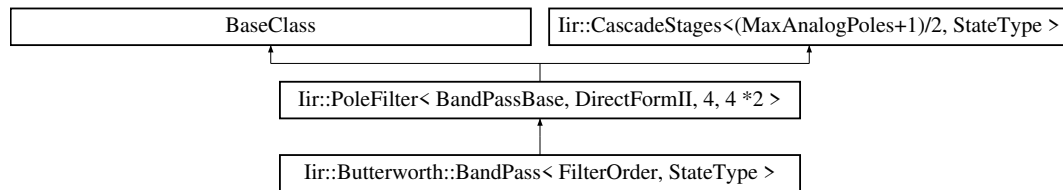
The documentation for this class was generated from the following files:

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

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

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

Inheritance diagram for `lir::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 `lir::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 lir::Butterworth::BandPass< FilterOrder, StateType >
```

[Butterworth](#) Bandpass filter.

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order <i>FilterOrder</i> |
| <i>StateType</i> | The filter topology: DirectFormI , DirectFormII , ... |

7.8.2 Member Function Documentation

[setup\(\)](#) [1/2]

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

Calculates the coefficients with the filter order provided by the instantiation

Parameters

| | |
|------------------------|----------------------------------|
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Centre frequency of the bandpass |
| <i>widthFrequency</i> | 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

| | |
|------------------------|--|
| <i>reqOrder</i> | The actual order which can be less than the instantiated one |
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Centre frequency of the bandpass |
| <i>widthFrequency</i> | 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

| | |
|------------------------|--|
| <i>centerFrequency</i> | Normalised centre frequency (0..1/2) of the bandpass |
| <i>widthFrequency</i> | 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

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

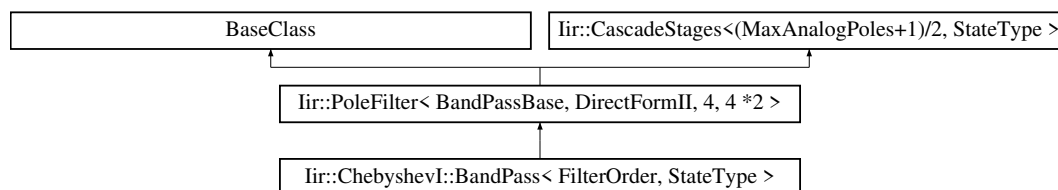
The documentation for this struct was generated from the following file:

- iir/Butterworth.h

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

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

Inheritance diagram for `lir::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 `lir::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 lir::ChebyshevI::BandPass< FilterOrder, StateType >
```

[ChebyshevI](#) bandpass filter

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order <i>FilterOrder</i> |
| <i>StateType</i> | The filter topology: DirectFormI , DirectFormII , ... |

7.9.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void lir::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

| | |
|------------------------|----------------------------------|
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Center frequency of the bandpass |
| <i>widthFrequency</i> | Frequency with of the passband |

Parameters

| | |
|-----------------|---|
| <i>rippleDb</i> | 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

| | |
|------------------------|--|
| <i>reqOrder</i> | Actual order for the filter calculations |
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Center frequency of the bandpass |
| <i>widthFrequency</i> | Frequency with of the passband |
| <i>rippleDb</i> | 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

| | |
|------------------------|--|
| <i>centerFrequency</i> | Normalised center frequency (0..1/2) of the bandpass |
| <i>widthFrequency</i> | Frequency with of the passband |
| <i>rippleDb</i> | 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

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

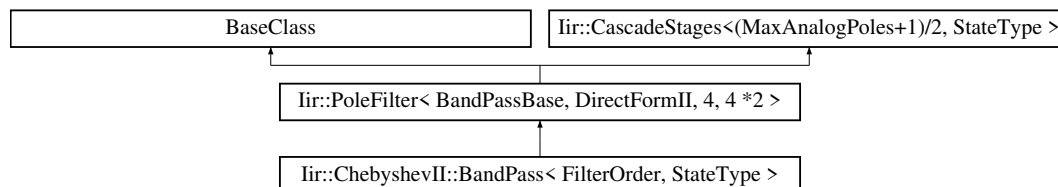
The documentation for this struct was generated from the following file:

- iir/ChebyshevI.h

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

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

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



Public Member Functions

- void [setup](#) (double sampleRate, double centerFrequency, double widthFrequency, double stopBandDb)
- void [setup](#) (int reqOrder, double sampleRate, double centerFrequency, double widthFrequency, double stopBandDb)
- 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

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order FilterOrder |
| <i>StateType</i> | 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

| | |
|------------------------|---|
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Center frequency of the bandpass |
| <i>widthFrequency</i> | Width of the bandpass |
| <i>stopBandDb</i> | 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

| | |
|------------------------|---|
| <i>reqOrder</i> | Requested order which can be less than the instantiated one |
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Center frequency of the bandpass |
| <i>widthFrequency</i> | Width of the bandpass |
| <i>stopBandDb</i> | 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

| | |
|------------------------|--|
| <i>centerFrequency</i> | Normalised centre frequency (0..1/2) of the bandpass |
| <i>widthFrequency</i> | Width of the bandpass |
| <i>stopBandDb</i> | 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

| | |
|-----------------|---|
| <i>reqOrder</i> | Requested order which can be less than the instantiated one |
|-----------------|---|

Parameters

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

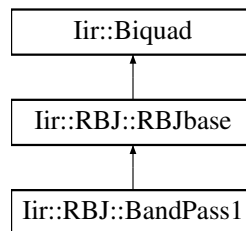
The documentation for this struct was generated from the following file:

- `iir/ChebyshevII.h`

7.11 `lir::RBJ::BandPass1` Struct Reference

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

Inheritance diagram for `lir::RBJ::BandPass1`:



Public Member Functions

- void [setupN](#) (double centerFrequency, double bandWidth)
- void [setup](#) (double sampleRate, double centerFrequency, double bandWidth)

Public Member Functions inherited from `lir::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 `lir::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

| | |
|------------------------|----------------------------------|
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Center frequency of the bandpass |
| <i>bandWidth</i> | Bandwidth in octaves |

setupN()

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

Calculates the coefficients

Parameters

| | |
|------------------------|----------------------------------|
| <i>centerFrequency</i> | Center frequency of the bandpass |
| <i>bandWidth</i> | Bandwidth in octaves |

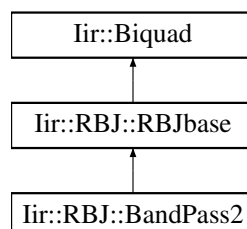
The documentation for this struct was generated from the following files:

- 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 [lir::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 [lir::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

| | |
|------------------------|----------------------------------|
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Center frequency of the bandpass |
| <i>bandWidth</i> | Bandwidth in octaves |

setupN()

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

Calculates the coefficients

Parameters

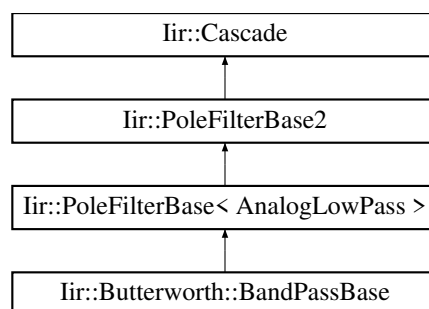
| | |
|------------------------|---|
| <i>centerFrequency</i> | Normalised centre frequency of the bandpass |
| <i>bandWidth</i> | Bandwidth in octaves |

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:



Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

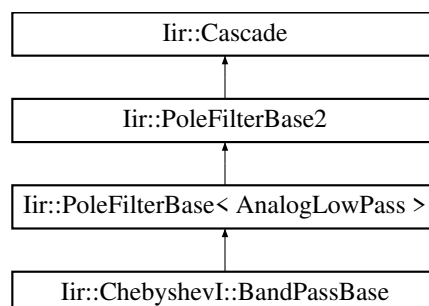
- int [getNumStages](#) () const
- const [Biquad](#) & [operator\[\]](#) (int index)
- complex_t [response](#) (double normalizedFrequency) const
- std::vector< [PoleZeroPair](#) > [getPoleZeros](#) () const

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:



Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

- int [getNumStages](#) () const

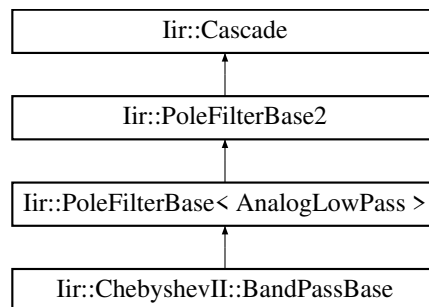
- const [Biquad](#) & [operator\[\]](#) (int index)
- complex_t [response](#) (double normalizedFrequency) const
- std::vector< [PoleZeroPair](#) > [getPoleZeros](#) () const

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:



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

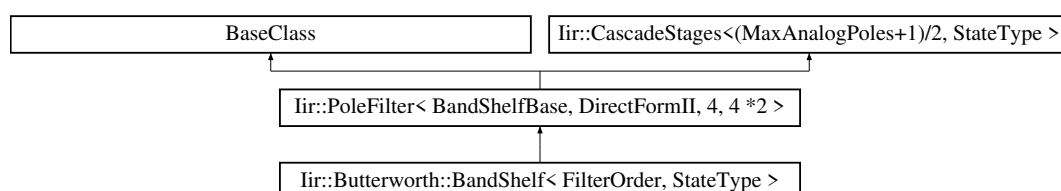
The documentation for this class was generated from the following files:

- 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 >:



Public Member Functions

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

Public Member Functions inherited from `lir::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 `lir::Butterworth::BandShelf< FilterOrder, StateType >`

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

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order <code>FilterOrder</code> |
| <i>StateType</i> | The filter topology: <code>DirectFormI</code> , <code>DirectFormII</code> , ... |

7.17.2 Member Function Documentation

`setup()` [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void lir::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

| | |
|------------------------|----------------------------------|
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Centre frequency of the passband |
| <i>widthFrequency</i> | Width of the passband |
| <i>gainDb</i> | The gain in the passband |

`setup()` [2/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void lir::Butterworth::BandShelf< FilterOrder, StateType >::setup (
    int reqOrder,
    double sampleRate,
    double centerFrequency,
    double widthFrequency,
```

```
double gainDb ) [inline]
```

Calculates the coefficients

Parameters

| | |
|------------------------|--|
| <i>reqOrder</i> | The actual order which can be less than the instantiated one |
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Centre frequency of the passband |
| <i>widthFrequency</i> | Width of the passband |
| <i>gainDb</i> | 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

| | |
|------------------------|--|
| <i>centerFrequency</i> | Normalised centre frequency (0..1/2) of the passband |
| <i>widthFrequency</i> | Width of the passband |
| <i>gainDb</i> | 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

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

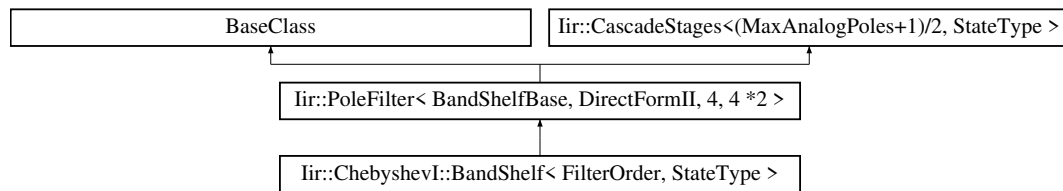
The documentation for this struct was generated from the following file:

- iir/Butterworth.h

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

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

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



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 `gainDb`, 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 `lir::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 lir::ChebyshevI::BandShelf< FilterOrder, StateType >`

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

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order <code>FilterOrder</code> |
| <i>StateType</i> | The filter topology: <code>DirectFormI</code> , <code>DirectFormII</code> , ... |

7.18.2 Member Function Documentation

`setup()` [1/2]

```

template<int FilterOrder = 4, class StateType = DirectFormII>
void lir::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

| | |
|------------------------|--|
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Center frequency of the passband |
| <i>widthFrequency</i> | Width of the passband. |
| <i>gainDb</i> | Gain in the passband. The stopband has 0 dB. |
| <i>rippleDb</i> | 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

| | |
|------------------------|--|
| <i>reqOrder</i> | Actual order for the filter calculations |
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Center frequency of the passband |
| <i>widthFrequency</i> | Width of the passband. |
| <i>gainDb</i> | Gain in the passband. The stopband has 0 dB. |
| <i>rippleDb</i> | 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

| | |
|------------------------|--|
| <i>centerFrequency</i> | Normalised centre frequency (0..1/2) of the passband |
| <i>widthFrequency</i> | Width of the passband. |
| <i>gainDb</i> | Gain in the passband. The stopband has 0 dB. |
| <i>rippleDb</i> | 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

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

Parameters

| | |
|-----------------|--|
| <i>rippleDb</i> | Permitted ripples in dB in the passband. |
|-----------------|--|

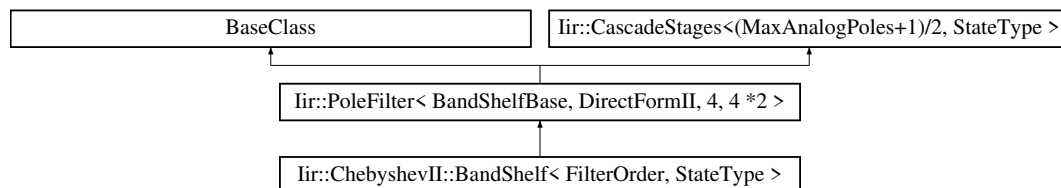
The documentation for this struct was generated from the following file:

- `iir/ChebyshevI.h`

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

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

Inheritance diagram for `lir::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 `lir::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 lir::ChebyshevII::BandShelf< FilterOrder, StateType >
```

[ChebyshevII](#) bandshelf filter. Bandpass with specified gain and 0 dB gain in the stopband.

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order FilterOrder |
| <i>StateType</i> | The filter topology: DirectFormI , DirectFormII , ... |

7.19.2 Member Function Documentation

setup() [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void lir::ChebyshevII::BandShelf< FilterOrder, StateType >::setup (
    double sampleRate,
```

```
double centerFrequency,
double widthFrequency,
double gainDb,
double stopBandDb ) [inline]
```

Calculates the coefficients of the filter

Parameters

| | |
|------------------------|--|
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Center frequency of the bandpass |
| <i>widthFrequency</i> | Width of the bandpass |
| <i>gainDb</i> | Gain in the passband. The stopband has always 0dB. |
| <i>stopBandDb</i> | 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

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

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

setupN() [2/2]

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

Calculates the coefficients of the filter

Parameters

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

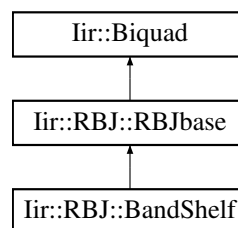
The documentation for this struct was generated from the following file:

- `iir/ChebyshevII.h`

7.20 `lir::RBJ::BandShelf` Struct Reference

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

Inheritance diagram for `lir::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 `lir::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 `lir::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

| | |
|------------------------|----------------------|
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | frequency |
| <i>gainDb</i> | Gain in the passband |
| <i>bandWidth</i> | Bandwidth in octaves |

setupN()

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

Calculates the coefficients

Parameters

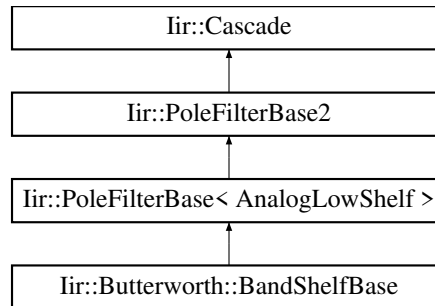
| | |
|------------------------|-----------------------------|
| <i>centerFrequency</i> | Normalised centre frequency |
| <i>gainDb</i> | Gain in the passband |
| <i>bandWidth</i> | Bandwidth in octaves |

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

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

7.21 `lir::Butterworth::BandShelfBase` Struct Reference

Inheritance diagram for `lir::Butterworth::BandShelfBase`:



Additional Inherited Members

Public Member Functions inherited from `lir::Cascade`

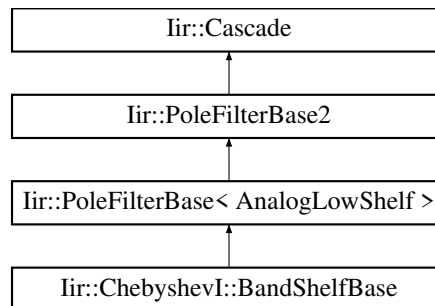
- `int` [`getNumStages`](#) () const
- `const Biquad & operator[]` (int index)
- `complex_t` [`response`](#) (double normalizedFrequency) const
- `std::vector< PoleZeroPair >` [`getPoleZeros`](#) () const

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

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

7.22 `lir::ChebyshevI::BandShelfBase` Struct Reference

Inheritance diagram for `lir::ChebyshevI::BandShelfBase`:



Additional Inherited Members

Public Member Functions inherited from `lir::Cascade`

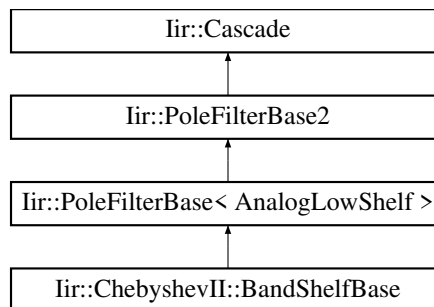
- `int` [`getNumStages`](#) () const
- `const Biquad & operator[]` (int index)
- `complex_t` [`response`](#) (double normalizedFrequency) const
- `std::vector< PoleZeroPair >` [`getPoleZeros`](#) () const

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:



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

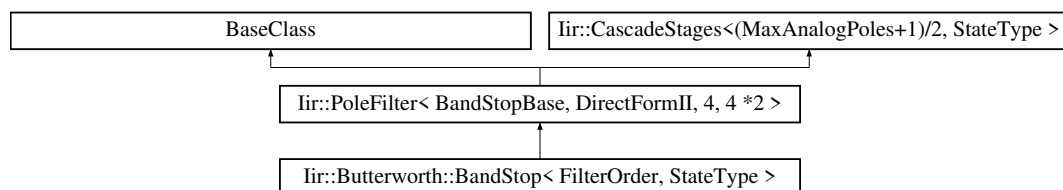
The documentation for this struct was generated from the following files:

- 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 >:



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 `lir::Butterworth::BandStop< FilterOrder, StateType >`

[Butterworth](#) Bandstop filter.

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order <code>FilterOrder</code> |
| <i>StateType</i> | The filter topology: DirectFormI , DirectFormII , ... |

7.24.2 Member Function Documentation

setup() [1/2]

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

Calculates the coefficients with the filter order provided by the instantiation

Parameters

| | |
|------------------------|----------------------------------|
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Centre frequency of the bandstop |
| <i>widthFrequency</i> | Width of the bandstop |

setup() [2/2]

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

Calculates the coefficients

Parameters

| | |
|------------------------|--|
| <i>reqOrder</i> | The actual order which can be less than the instantiated one |
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Centre frequency of the bandstop |
| <i>widthFrequency</i> | Width of the bandstop |

setupN() [1/2]

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

Calculates the coefficients with the filter order provided by the instantiation

Parameters

| | |
|------------------------|--|
| <i>centerFrequency</i> | Normalised centre frequency (0..1/2) of the bandstop |
| <i>widthFrequency</i> | 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

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

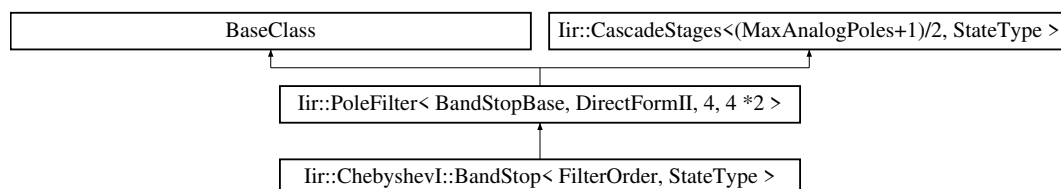
The documentation for this struct was generated from the following file:

- iir/Butterworth.h

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

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

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

**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

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order <i>FilterOrder</i> |
| <i>StateType</i> | 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

| | |
|------------------------|---|
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Center frequency of the notch |
| <i>widthFrequency</i> | Frequency with of the notch |
| <i>rippleDb</i> | 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

| | |
|------------------------|--|
| <i>reqOrder</i> | Actual order for the filter calculations |
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Center frequency of the notch |
| <i>widthFrequency</i> | Frequency with of the notch |
| <i>rippleDb</i> | 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

| | |
|------------------------|---|
| <i>centerFrequency</i> | Normalised centre frequency (0..1/2) of the notch |
| <i>widthFrequency</i> | Frequency width of the notch |
| <i>rippleDb</i> | 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

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

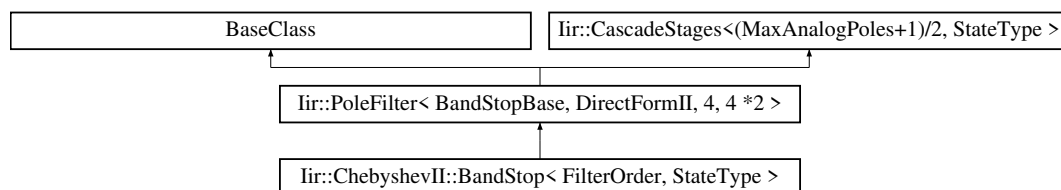
The documentation for this struct was generated from the following file:

- iir/ChebyshevI.h

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

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

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



Public Member Functions

- void [setup](#) (double sampleRate, double centerFrequency, double widthFrequency, double stopBandDb)
- void [setup](#) (int reqOrder, double sampleRate, double centerFrequency, double widthFrequency, double stopBandDb)
- 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 lir::ChebyshevII::BandStop< FilterOrder, StateType >`

`ChebyshevII` bandstop filter.

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order <code>FilterOrder</code> |
| <i>StateType</i> | The filter topology: <code>DirectFormI</code> , <code>DirectFormII</code> , ... |

7.26.2 Member Function Documentation

`setup()` [1/2]

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

Calculates the coefficients of the filter

Parameters

| | |
|------------------------|---|
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Center frequency of the bandstop |
| <i>widthFrequency</i> | Width of the bandstop |
| <i>stopBandDb</i> | Permitted ripples in dB in the stopband |

`setup()` [2/2]

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

Calculates the coefficients of the filter

Parameters

| | |
|------------------------|---|
| <i>reqOrder</i> | Requested order which can be less than the instantiated one |
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Center frequency of the bandstop |
| <i>widthFrequency</i> | Width of the bandstop |
| <i>stopBandDb</i> | 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

| | |
|------------------------|--|
| <i>centerFrequency</i> | Normalised centre frequency (0..1/2) of the bandstop |
| <i>widthFrequency</i> | Width of the bandstop |
| <i>stopBandDb</i> | 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

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

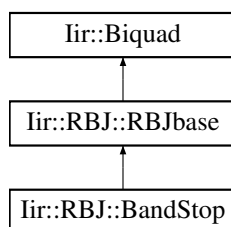
The documentation for this struct was generated from the following file:

- 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

| | |
|------------------------|----------------------------------|
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Center frequency of the bandstop |
| <i>bandWidth</i> | Bandwidth in octaves |

setupN()

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

Calculates the coefficients

Parameters

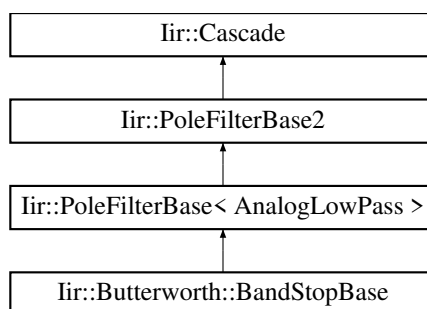
| | |
|------------------------|---|
| <i>centerFrequency</i> | Normalised Centre frequency of the bandstop |
| <i>bandWidth</i> | Bandwidth in octaves |

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:



Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

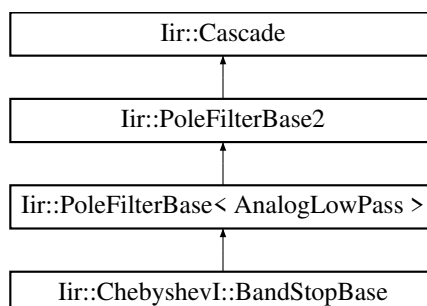
- int [getNumStages](#) () const
- const [Biquad](#) & [operator\[\]](#) (int index)
- complex_t [response](#) (double normalizedFrequency) const
- std::vector< [PoleZeroPair](#) > [getPoleZeros](#) () const

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:



Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

- int [getNumStages](#) () const

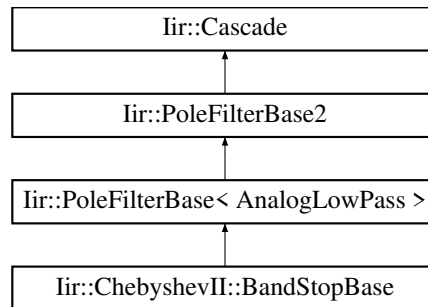
- const [Biquad](#) & [operator\[\]](#) (int index)
- complex_t [response](#) (double normalizedFrequency) const
- std::vector< [PoleZeroPair](#) > [getPoleZeros](#) () const

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:



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

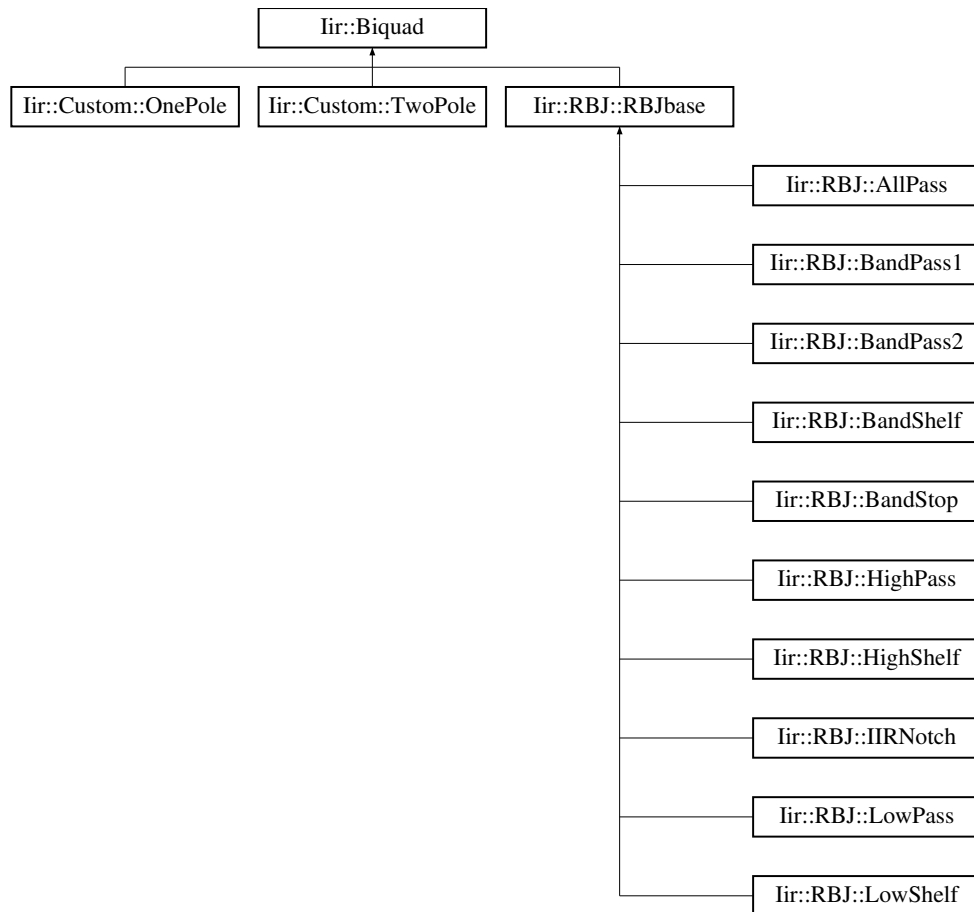
low pass to band stop transform

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

| | |
|--------------|--|
| <i>scale</i> | Multplies 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

| | |
|--------------|---|
| <i>s</i> | The sample to be filtered. |
| <i>state</i> | 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

| | |
|----------------------------|---------------------------------|
| <i>normalizedFrequency</i> | 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

| | |
|-----------|---------------------|
| <i>a0</i> | 1st IIR coefficient |
| <i>a1</i> | 2nd IIR coefficient |
| <i>a2</i> | 3rd IIR coefficient |
| <i>b0</i> | 1st FIR coefficient |
| <i>b1</i> | 2nd FIR coefficient |
| <i>b2</i> | 3rd FIR coefficient |

setIdentity()

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

Sets the coefficients 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.

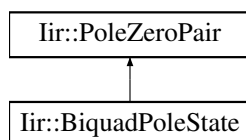
The documentation for this class was generated from the following files:

- 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

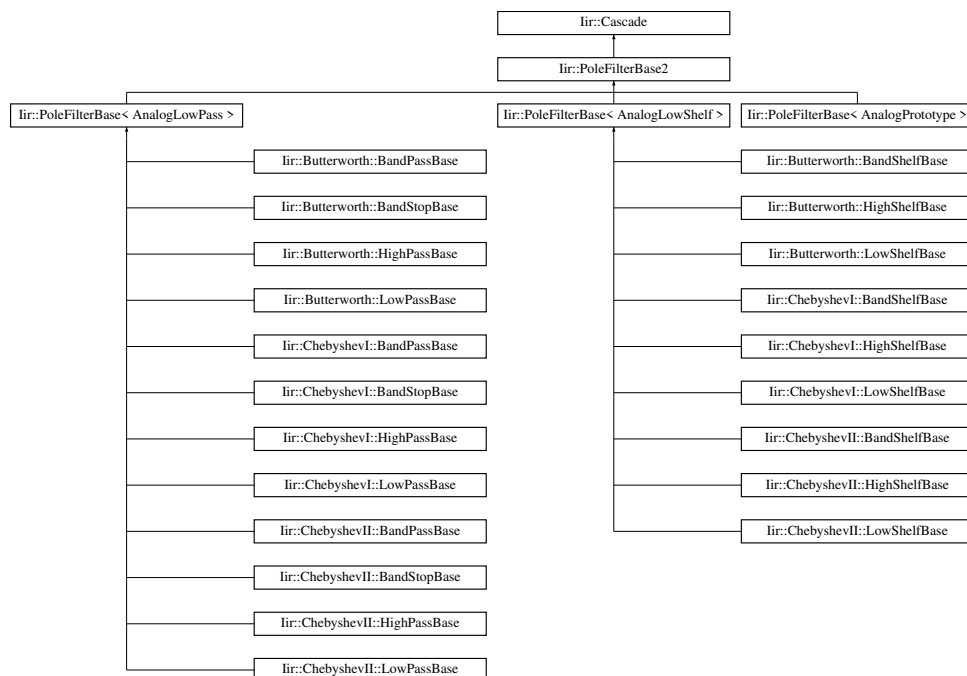
Expresses a biquad as a pair of pole/zeros, with gain values so that the coefficients can be reconstructed precisely. The documentation for this struct was generated from the following files:

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

7.34 Iir::Cascade Class Reference

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

Inheritance diagram for Iir::Cascade:



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 Biquad 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

| | |
|----------------------------|-----------------------------------|
| <i>normalizedFrequency</i> | Frequency from 0 to 0.5 (Nyquist) |
|----------------------------|-----------------------------------|

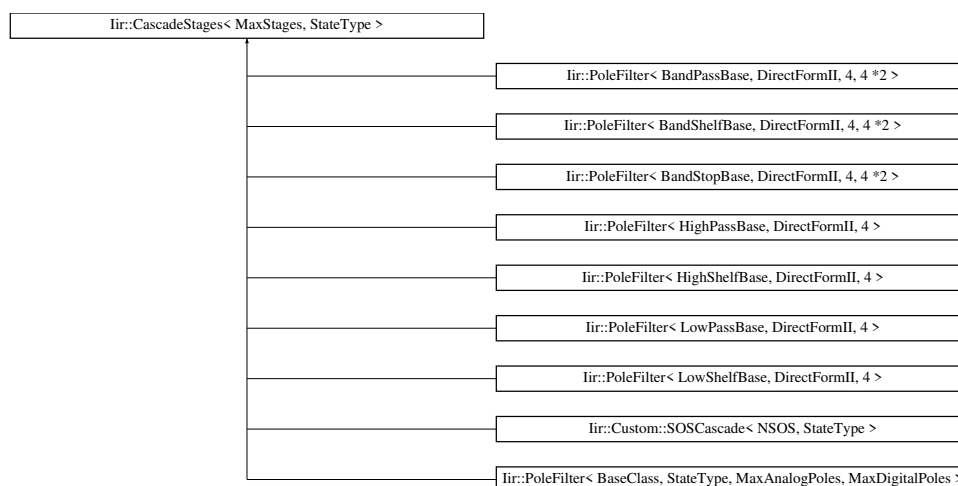
The documentation for this class was generated from the following files:

- 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 >`:



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 `lir::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 lir::CascadeStages< MaxStages, StateType >::filter (
    const Sample in ) [inline]
```

Filters one sample through the whole chain of biquads and return the result

Parameters

| | |
|-----------|-----------------------|
| <i>in</i> | Sample to be filtered |
|-----------|-----------------------|

Returns

filtered sample

getCascadeStorage()

```
template<int MaxStages, class StateType >
const Cascade::Storage lir::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

| | |
|------------------------|--|
| <i>sosCoefficients</i> | 2D array in Python style sos ordering: 0-2: FIR, 3-5: IIR coeff. |
|------------------------|--|

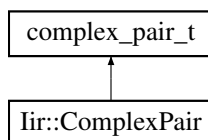
The documentation for this class was generated from the following file:

- 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 `lir::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:

$$v[n] = x[n] - (a1/a0)*v[n-1] - (a2/a0)*v[n-2] \quad y(n) = (b0/a0)*v[n] + (b1/a0)*v[n-1] + (b2/a0)*v[n-2]$$

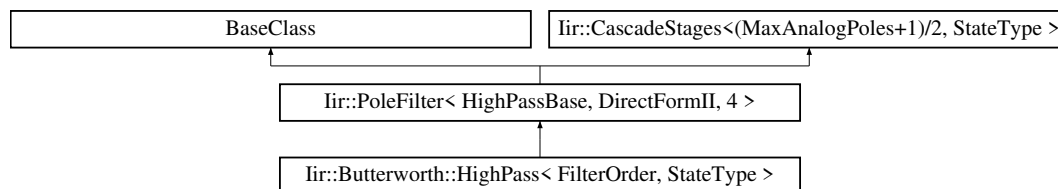
The documentation for this class was generated from the following file:

- `iir/State.h`

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

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

Inheritance diagram for `lir::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 `lir::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 lir::Butterworth::HighPass< FilterOrder, StateType >
```

[Butterworth](#) Highpass filter.

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order <code>FilterOrder</code> |
| <i>StateType</i> | 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

| | |
|------------------------|------------------|
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | 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

| | |
|------------------------|--|
| <i>reqOrder</i> | The actual order which can be less than the instantiated one |
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | 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

| | |
|------------------------|--------------------------------------|
| <i>cutoffFrequency</i> | 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

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

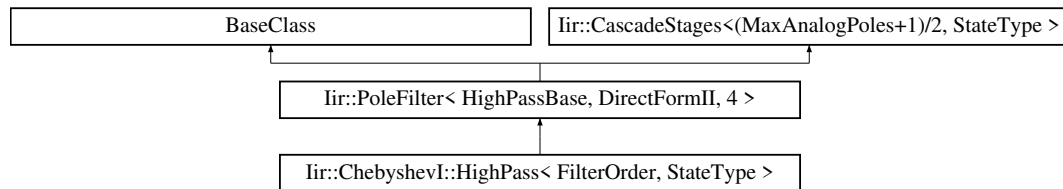
The documentation for this struct was generated from the following file:

- `iir/Butterworth.h`

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

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

Inheritance diagram for `lir::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 `lir::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 lir::ChebyshevI::HighPass< FilterOrder, StateType >
```

[ChebyshevI](#) highpass filter

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order <code>FilterOrder</code> |
| <i>StateType</i> | 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

| | |
|------------------------|---|
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency. |
| <i>rippleDb</i> | 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

| | |
|------------------------|--|
| <i>reqOrder</i> | Actual order for the filter calculations |
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency. |
| <i>rippleDb</i> | 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

| | |
|------------------------|---|
| <i>cutoffFrequency</i> | Normalised cutoff frequency (0..1/2) |
| <i>rippleDb</i> | 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

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

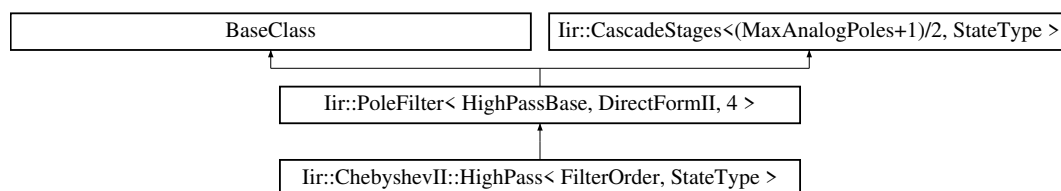
The documentation for this struct was generated from the following file:

- iir/ChebyshevI.h

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

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

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



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 `lir::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 lir::ChebyshevII::HighPass< FilterOrder, StateType >`

[ChebyshevII](#) highpass filter

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order FilterOrder |
| <i>StateType</i> | The filter topology: DirectFormI , DirectFormII , ... |

7.41.2 Member Function Documentation

[setup\(\)](#) [1/2]

```

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

Calculates the coefficients of the filter

Parameters

| | |
|------------------------|---|
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency. |
| <i>stopBandDb</i> | Permitted ripples in dB in the stopband |

[setup\(\)](#) [2/2]

```

template<int FilterOrder = 4, class StateType = DirectFormII>
void lir::ChebyshevII::HighPass< FilterOrder, StateType >::setup (
  
```

```

    int reqOrder,
    double sampleRate,
    double cutoffFrequency,
    double stopBandDb ) [inline]

```

Calculates the coefficients of the filter

Parameters

| | |
|------------------------|---|
| <i>reqOrder</i> | Requested order which can be less than the instantiated one |
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency. |
| <i>stopBandDb</i> | 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

| | |
|------------------------|---|
| <i>cutoffFrequency</i> | Normalised cutoff frequency (0..1/2) |
| <i>stopBandDb</i> | 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

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

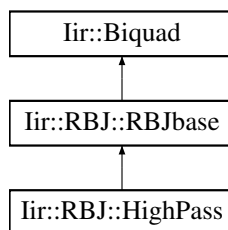
The documentation for this struct was generated from the following file:

- 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

| | |
|------------------------|---|
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency |
| <i>q</i> | 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

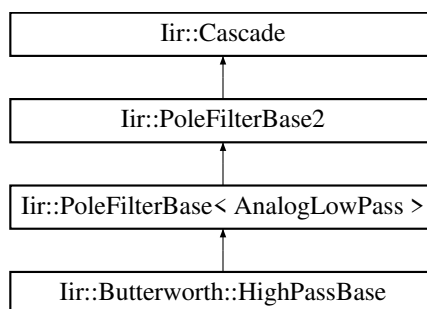
| | |
|------------------------|---|
| <i>cutoffFrequency</i> | Normalised cutoff frequency (0..1/2) |
| <i>q</i> | Q factor determines the resonance peak at the cutoff. |

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:

**Additional Inherited Members****Public Member Functions inherited from Iir::Cascade**

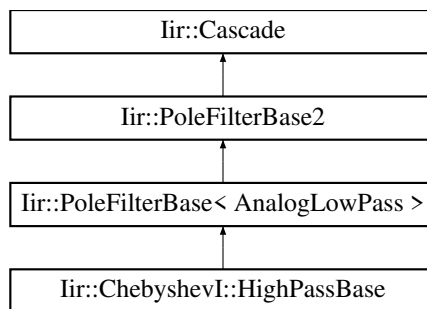
- int [getNumStages](#) () const
- const [Biquad](#) & [operator\[\]](#) (int index)
- complex_t [response](#) (double normalizedFrequency) const
- std::vector< [PoleZeroPair](#) > [getPoleZeros](#) () const

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:



Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

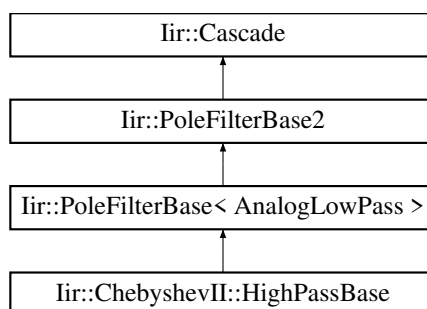
- int [getNumStages](#) () const
- const [Biquad](#) & [operator\[\]](#) (int index)
- complex_t [response](#) (double normalizedFrequency) const
- std::vector< [PoleZeroPair](#) > [getPoleZeros](#) () const

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:



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

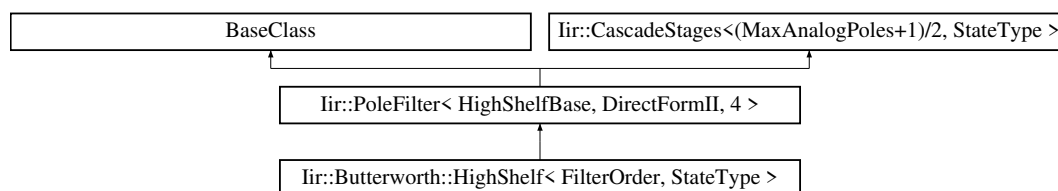
The documentation for this class was generated from the following files:

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

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

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

Inheritance diagram for `lir::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 `lir::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 lir::Butterworth::HighShelf< FilterOrder, StateType >
```

[Butterworth](#) high shelf filter. Above the cutoff the filter has a specified gain and below it has 0 dB.

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order <i>FilterOrder</i> |
| <i>StateType</i> | The filter topology: DirectFormI , DirectFormII , ... |

7.47.2 Member Function Documentation

[setup\(\)](#) [1/2]

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

Calculates the coefficients with the filter order provided by the instantiation

Parameters

| | |
|------------------------|--|
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff |
| <i>gainDb</i> | 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

| | |
|------------------------|--|
| <i>reqOrder</i> | The actual order which can be less than the instantiated one |
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff |
| <i>gainDb</i> | 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

| | |
|------------------------|--|
| <i>cutoffFrequency</i> | Normalised cutoff frequency (0..1/2) |
| <i>gainDb</i> | 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

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

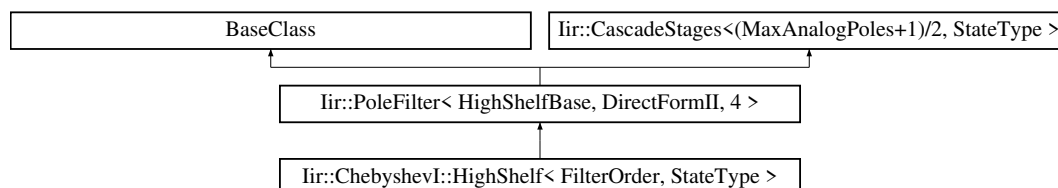
The documentation for this struct was generated from the following file:

- iir/Butterworth.h

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

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

Inheritance diagram for `lir::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 `lir::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 lir::ChebyshevI::HighShelf< FilterOrder, StateType >
```

[ChebyshevI](#) high shelf filter. Specified gain in the passband. Otherwise 0 dB.

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order <code>FilterOrder</code> |
| <i>StateType</i> | 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

| | |
|------------------------|---|
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency. |
| <i>gainDb</i> | Gain in the passband |
| <i>rippleDb</i> | 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

| | |
|------------------------|--|
| <i>reqOrder</i> | Actual order for the filter calculations |
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency. |
| <i>gainDb</i> | Gain in the passband |
| <i>rippleDb</i> | 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

| | |
|------------------------|---|
| <i>cutoffFrequency</i> | Normalised cutoff frequency (0..1/2) |
| <i>gainDb</i> | Gain in the passband |
| <i>rippleDb</i> | 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

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

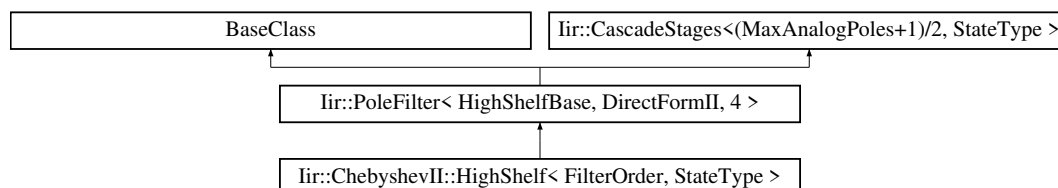
The documentation for this struct was generated from the following file:

- `iir/ChebyshevI.h`

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

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

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



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

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order FilterOrder |
| <i>StateType</i> | 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

| | |
|------------------------|--|
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency. |
| <i>gainDb</i> | Gain the passband. The stopband has 0 dB gain. |
| <i>stopBandDb</i> | 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

| | |
|------------------------|---|
| <i>reqOrder</i> | Requested order which can be less than the instantiated one |
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency. |
| <i>gainDb</i> | Gain the passband. The stopband has 0 dB gain. |
| <i>stopBandDb</i> | 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

| | |
|------------------------|--|
| <i>cutoffFrequency</i> | Normalised cutoff frequency (0..1/2) |
| <i>gainDb</i> | Gain the passband. The stopband has 0 dB gain. |
| <i>stopBandDb</i> | 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

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

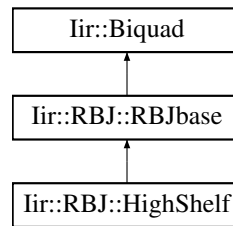
The documentation for this struct was generated from the following file:

- iir/ChebyshevII.h

7.50 lir::RBJ::HighShelf Struct Reference

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

Inheritance diagram for lir::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 [lir::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 [lir::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

| | |
|------------------------|--|
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency |
| <i>gainDb</i> | Gain in the passband |
| <i>shelfSlope</i> | 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

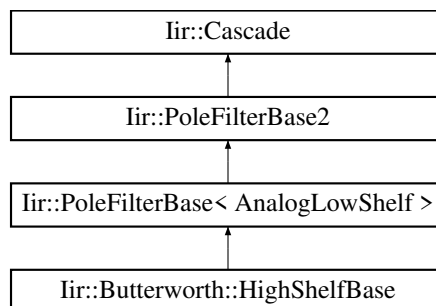
| | |
|------------------------|--|
| <i>cutoffFrequency</i> | Normalised cutoff frequency |
| <i>gainDb</i> | Gain in the passband |
| <i>shelfSlope</i> | Slope between stop/passband. 1 = as steep as it can. |

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:



Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

- int [getNumStages](#) () const

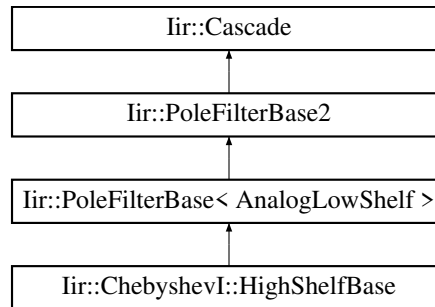
- const [Biquad](#) & [operator\[\]](#) (int index)
- complex_t [response](#) (double normalizedFrequency) const
- std::vector< [PoleZeroPair](#) > [getPoleZeros](#) () const

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:



Additional Inherited Members

Public Member Functions inherited from [Iir::Cascade](#)

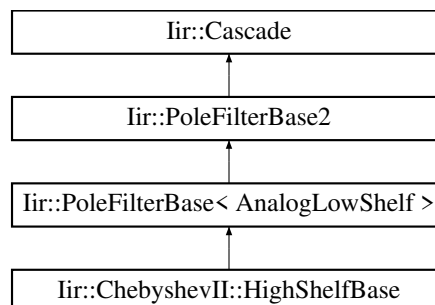
- int [getNumStages](#) () const
- const [Biquad](#) & [operator\[\]](#) (int index)
- complex_t [response](#) (double normalizedFrequency) const
- std::vector< [PoleZeroPair](#) > [getPoleZeros](#) () const

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:



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

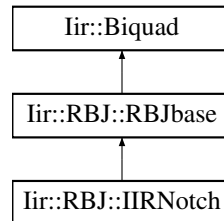
The documentation for this struct was generated from the following files:

- `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 $r = \exp(-\pi * (\text{centerFrequency}/\text{sampleRate})/q_factor)$ 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

| | |
|------------------------|----------------------------------|
| <i>sampleRate</i> | Sampling rate |
| <i>centerFrequency</i> | Center frequency of the notch |
| <i>q_factor</i> | Q factor of the notch (1 to ~20) |

setupN()

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

Calculates the coefficients

Parameters

| | |
|------------------------|--|
| <i>centerFrequency</i> | Normalised centre frequency of the notch |
| <i>q_factor</i> | 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](#)

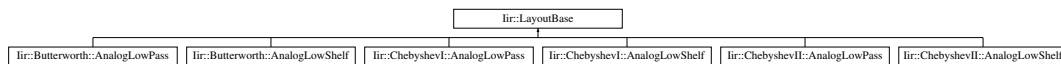
The documentation for this class was generated from the following file:

- iir/Layout.h

7.56 Iir::LayoutBase Class Reference

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

Inheritance diagram for Iir::LayoutBase:



7.56.1 Detailed Description

Base uses pointers to reduce template instantiations

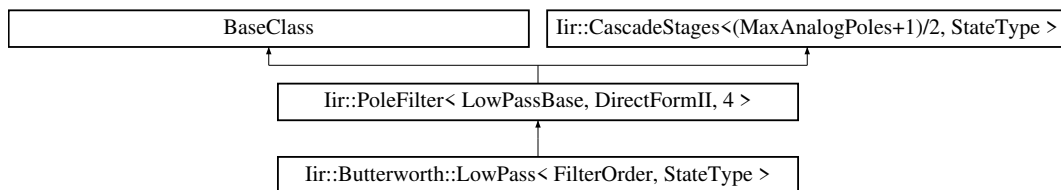
The documentation for this class was generated from the following file:

- `iir/Layout.h`

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

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

Inheritance diagram for `lir::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 `lir::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 lir::Butterworth::LowPass< FilterOrder, StateType >
```

`Butterworth` Lowpass filter.

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order <code>FilterOrder</code> |
| <i>StateType</i> | The filter topology: <code>DirectFormI</code> , <code>DirectFormII</code> , ... |

7.57.2 Member Function Documentation

`setup()` [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void lir::Butterworth::LowPass< FilterOrder, StateType >::setup (
    double sampleRate,
```

```
double cutoffFrequency ) [inline]
```

Calculates the coefficients

Parameters

| | |
|------------------------|---------------|
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | 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

| | |
|------------------------|--|
| <i>reqOrder</i> | The actual order which can be less than the instantiated one |
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | 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

| | |
|------------------------|--------------------------------------|
| <i>cutoffFrequency</i> | 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

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

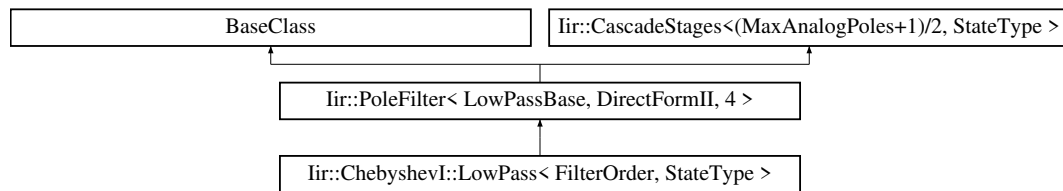
The documentation for this struct was generated from the following file:

- iir/Butterworth.h

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

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

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



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 `lir::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 lir::ChebyshevI::LowPass< FilterOrder, StateType >`

[ChebyshevI](#) lowpass filter

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order FilterOrder |
| <i>StateType</i> | The filter topology: DirectFormI , DirectFormII , ... |

7.58.2 Member Function Documentation

[setup\(\)](#) [1/2]

```

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

Calculates the coefficients of the filter at the order FilterOrder

Parameters

| | |
|------------------------|---|
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency |
| <i>rippleDb</i> | Permitted ripples in dB in the passband |

[setup\(\)](#) [2/2]

```

template<int FilterOrder = 4, class StateType = DirectFormII>
void lir::ChebyshevI::LowPass< FilterOrder, StateType >::setup (
  
```

```

    int reqOrder,
    double sampleRate,
    double cutoffFrequency,
    double rippleDb ) [inline]

```

Calculates the coefficients of the filter at specified order

Parameters

| | |
|------------------------|--|
| <i>reqOrder</i> | Actual order for the filter calculations |
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency. |
| <i>rippleDb</i> | 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

| | |
|------------------------|---|
| <i>cutoffFrequency</i> | Normalised cutoff frequency (0..1/2) |
| <i>rippleDb</i> | 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

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

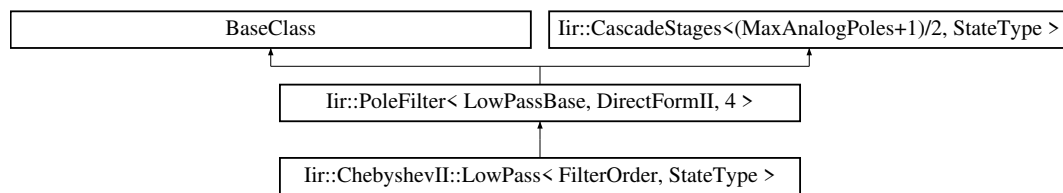
The documentation for this struct was generated from the following file:

- iir/ChebyshevI.h

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

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

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



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 `lir::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 lir::ChebyshevII::LowPass< FilterOrder, StateType >`

[ChebyshevII](#) lowpass filter

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order FilterOrder |
| <i>StateType</i> | The filter topology: DirectFormI , DirectFormII , ... |

7.59.2 Member Function Documentation

[setup\(\)](#) [1/2]

```

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

Calculates the coefficients of the filter

Parameters

| | |
|------------------------|---|
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency. |
| <i>stopBandDb</i> | Permitted ripples in dB in the stopband |

[setup\(\)](#) [2/2]

```

template<int FilterOrder = 4, class StateType = DirectFormII>
void lir::ChebyshevII::LowPass< FilterOrder, StateType >::setup (
  
```

```

    int reqOrder,
    double sampleRate,
    double cutoffFrequency,
    double stopBandDb ) [inline]

```

Calculates the coefficients of the filter

Parameters

| | |
|------------------------|---|
| <i>reqOrder</i> | Requested order which can be less than the instantiated one |
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency. |
| <i>stopBandDb</i> | 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

| | |
|------------------------|---|
| <i>cutoffFrequency</i> | Normalised cutoff frequency (0..1/2) |
| <i>stopBandDb</i> | 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

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

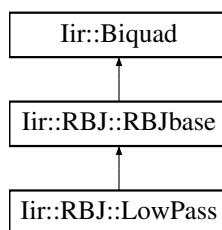
The documentation for this struct was generated from the following file:

- 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

| | |
|------------------------|---|
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency |
| <i>q</i> | 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

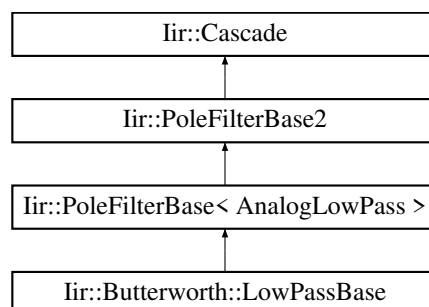
| | |
|------------------------|---|
| <i>cutoffFrequency</i> | Normalised cutoff frequency |
| <i>q</i> | Q factor determines the resonance peak at the cutoff. |

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:

**Additional Inherited Members****Public Member Functions inherited from Iir::Cascade**

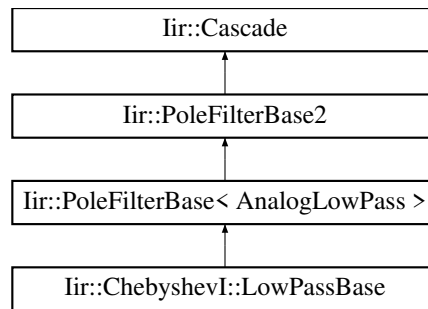
- int [getNumStages](#) () const
- const [Biquad](#) & [operator\[\]](#) (int index)
- complex_t [response](#) (double normalizedFrequency) const
- std::vector< [PoleZeroPair](#) > [getPoleZeros](#) () const

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:



Additional Inherited Members

Public Member Functions inherited from Iir::Cascade

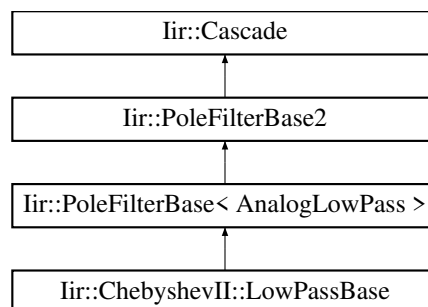
- int [getNumStages](#) () const
- const [Biquad](#) & [operator\[\]](#) (int index)
- complex_t [response](#) (double normalizedFrequency) const
- std::vector< [PoleZeroPair](#) > [getPoleZeros](#) () const

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:



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

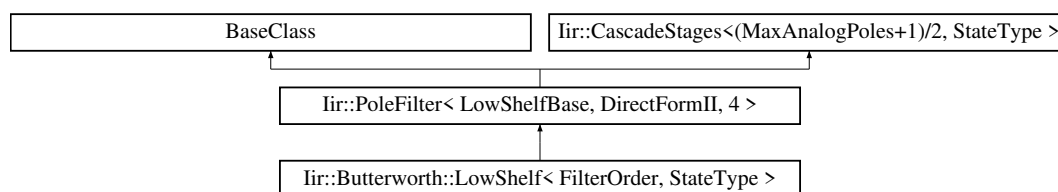
The documentation for this class was generated from the following files:

- 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 >`:



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

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order <i>FilterOrder</i> |
| <i>StateType</i> | 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

| | |
|------------------------|--|
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff |
| <i>gainDb</i> | 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

| | |
|------------------------|--|
| <i>reqOrder</i> | The actual order which can be less than the instantiated one |
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff |
| <i>gainDb</i> | 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

| | |
|------------------------|--|
| <i>cutoffFrequency</i> | Normalised cutoff frequency (0..1/2) |
| <i>gainDb</i> | 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

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

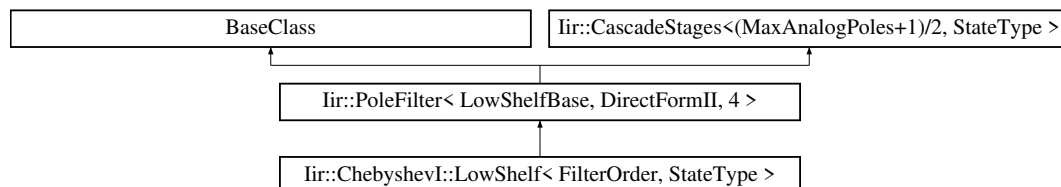
The documentation for this struct was generated from the following file:

- iir/Butterworth.h

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

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

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



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

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order FilterOrder |
| <i>StateType</i> | 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

| | |
|-------------------|---------------|
| <i>sampleRate</i> | Sampling rate |
|-------------------|---------------|

Parameters

| | |
|------------------------|---|
| <i>cutoffFrequency</i> | Cutoff frequency. |
| <i>gainDb</i> | Gain in the passband |
| <i>rippleDb</i> | 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

| | |
|------------------------|--|
| <i>reqOrder</i> | Actual order for the filter calculations |
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency. |
| <i>gainDb</i> | Gain in the passband |
| <i>rippleDb</i> | 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

| | |
|------------------------|---|
| <i>cutoffFrequency</i> | Normalised cutoff frequency (0..1/2) |
| <i>gainDb</i> | Gain in the passband |
| <i>rippleDb</i> | 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

| | |
|-----------------|--|
| <i>reqOrder</i> | Actual order for the filter calculations |
|-----------------|--|

Parameters

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

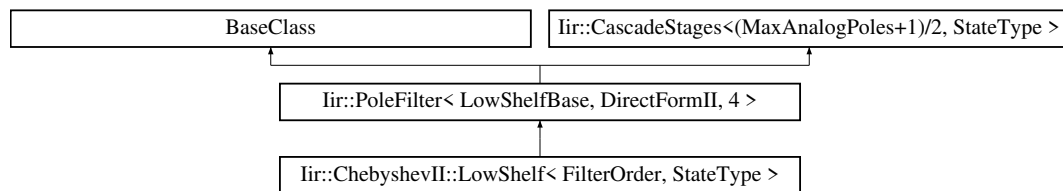
The documentation for this struct was generated from the following file:

- iir/ChebyshevI.h

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

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

Inheritance diagram for `lir::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 `lir::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 lir::ChebyshevII::LowShelf< FilterOrder, StateType >
```

[ChebyshevII](#) low shelf filter. Specified gain in the passband and 0dB in the stopband.

Parameters

| | |
|--------------------|---|
| <i>FilterOrder</i> | Reserves memory for a filter of the order <code>FilterOrder</code> |
| <i>StateType</i> | The filter topology: DirectFormI , DirectFormII , ... |

7.67.2 Member Function Documentation

`setup()` [1/2]

```
template<int FilterOrder = 4, class StateType = DirectFormII>
void lir::ChebyshevII::LowShelf< FilterOrder, StateType >::setup (
    double sampleRate,
```

```
double cutoffFrequency,
double gainDb,
double stopBandDb ) [inline]
```

Calculates the coefficients of the filter

Parameters

| | |
|------------------------|---|
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency. |
| <i>gainDb</i> | Gain of the passband. The stopband has 0 dB gain. |
| <i>stopBandDb</i> | 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

| | |
|------------------------|---|
| <i>reqOrder</i> | Requested order which can be less than the instantiated one |
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency |
| <i>gainDb</i> | Gain of the passband. The stopband has 0 dB gain. |
| <i>stopBandDb</i> | 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

| | |
|------------------------|---|
| <i>cutoffFrequency</i> | Normalised cutoff frequency (0..1/2) |
| <i>gainDb</i> | Gain of the passband. The stopband has 0 dB gain. |
| <i>stopBandDb</i> | 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

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

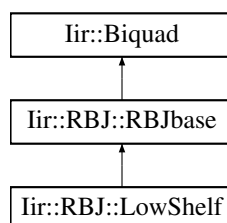
The documentation for this struct was generated from the following file:

- 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

| | |
|------------------------|--|
| <i>sampleRate</i> | Sampling rate |
| <i>cutoffFrequency</i> | Cutoff frequency |
| <i>gainDb</i> | Gain in the passband |
| <i>shelfSlope</i> | 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

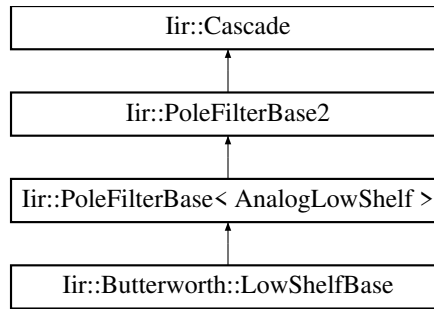
| | |
|------------------------|--|
| <i>cutoffFrequency</i> | Normalised cutoff frequency |
| <i>gainDb</i> | Gain in the passband |
| <i>shelfSlope</i> | Slope between stop/passband. 1 = as steep as it can. |

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:



Additional Inherited Members

Public Member Functions inherited from [Iir::Cascade](#)

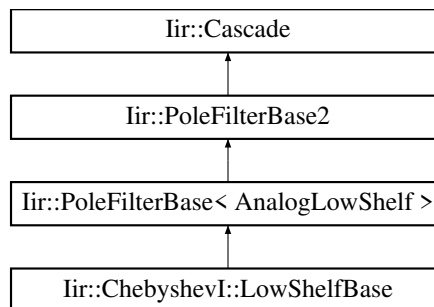
- int [getNumStages](#) () const
- const [Biquad](#) & [operator\[\]](#) (int index)
- complex_t [response](#) (double normalizedFrequency) const
- std::vector< [PoleZeroPair](#) > [getPoleZeros](#) () const

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:



Additional Inherited Members

Public Member Functions inherited from [Iir::Cascade](#)

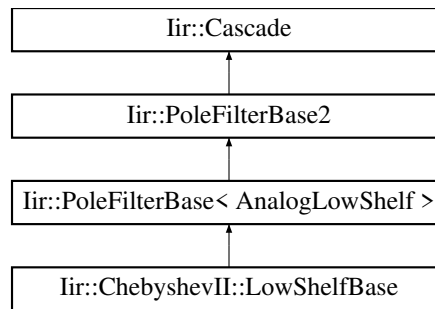
- int [getNumStages](#) () const
- const [Biquad](#) & [operator\[\]](#) (int index)
- complex_t [response](#) (double normalizedFrequency) const
- std::vector< [PoleZeroPair](#) > [getPoleZeros](#) () const

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:



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

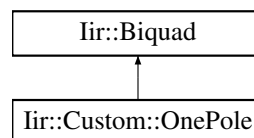
The documentation for this struct was generated from the following files:

- 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

| | |
|--------------|---|
| <i>scale</i> | Scale the FIR coefficients by this factor |
| <i>pole</i> | Position of the pole on the real axis |
| <i>zero</i> | Position of the zero on the real axis |

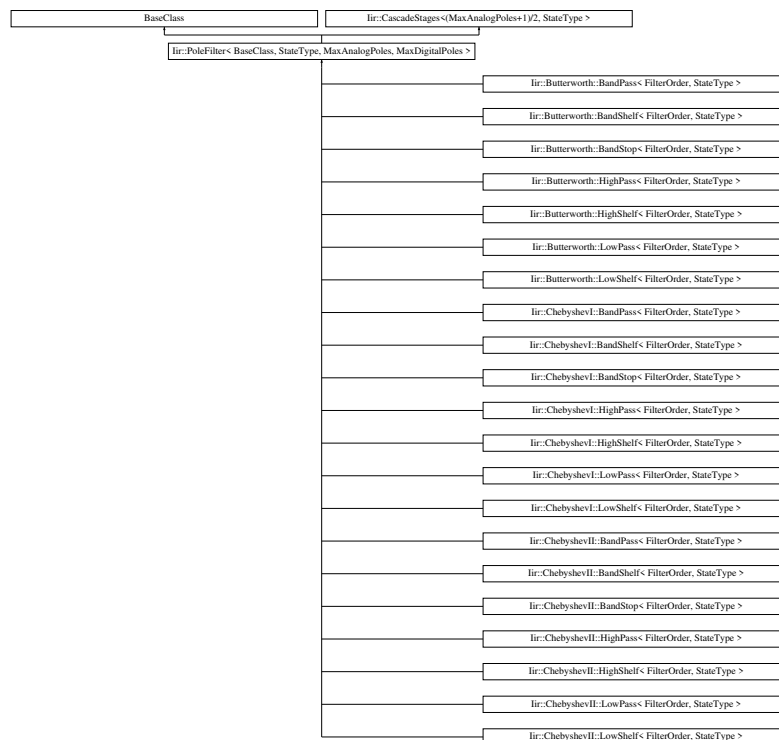
The documentation for this struct was generated from the following files:

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

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

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

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



Additional Inherited Members

Public Member Functions inherited from `lir::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 lir::PoleFilter< BaseClass, StateType, MaxAnalogPoles, MaxDigitalPoles >
```

Storage for pole filters

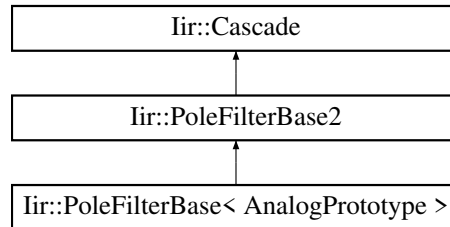
The documentation for this struct was generated from the following file:

- iir/PoleFilter.h

7.74 `lir::PoleFilterBase< AnalogPrototype >` Class Template Reference

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

Inheritance diagram for `lir::PoleFilterBase< AnalogPrototype >`:



Additional Inherited Members

Public Member Functions inherited from [lir::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 lir::PoleFilterBase< AnalogPrototype >
```

Serves a container to hold the analog prototype and the digital pole/zero layout.

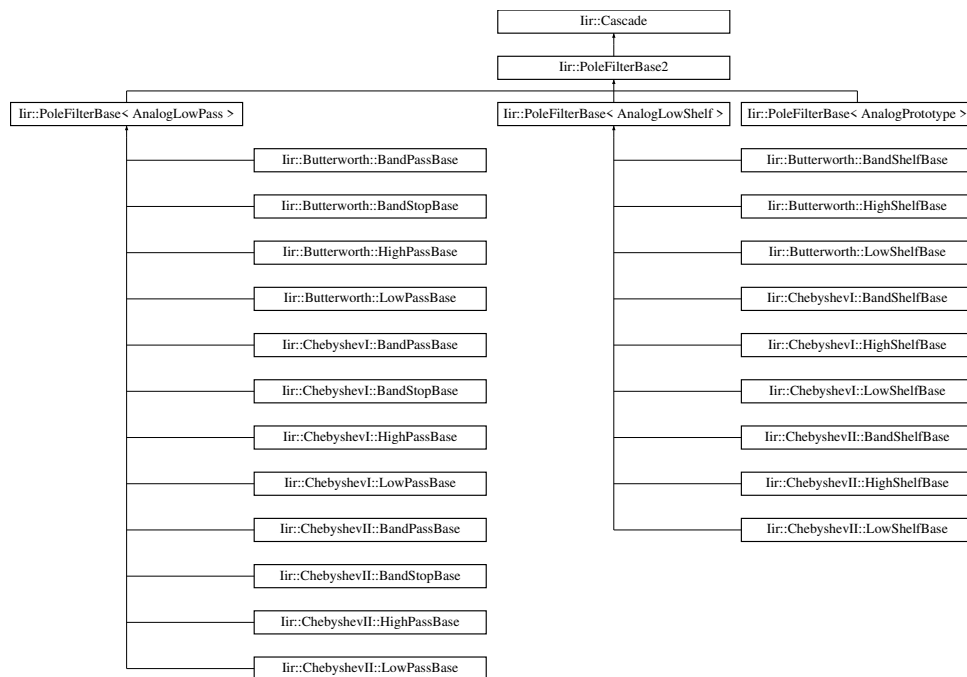
The documentation for this class was generated from the following file:

- iir/PoleFilter.h

7.75 `lir::PoleFilterBase2` Class Reference

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

Inheritance diagram for `lir::PoleFilterBase2`:



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

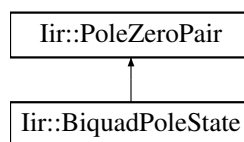
The documentation for this class was generated from the following file:

- 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)

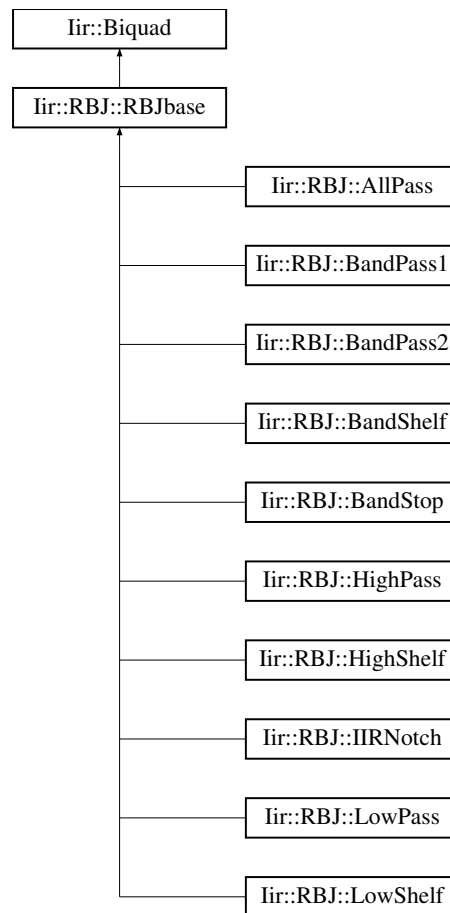
The documentation for this struct was generated from the following file:

- 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

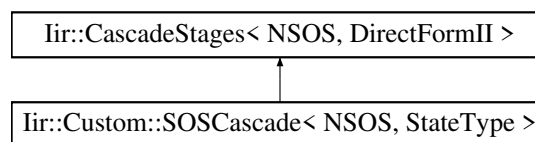
The documentation for this struct was generated from the following file:

- iir/RBJ.h

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

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

Inheritance diagram for Iir::Custom::SOSCascade< NSOS, StateType >:



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

| | |
|------------------|---|
| <i>NSOS</i> | The number of 2nd order filters / biquads. |
| <i>StateType</i> | 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

| | |
|------------------------|---|
| <i>sosCoefficients</i> | 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

| | |
|------------------------|---|
| <i>sosCoefficients</i> | 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

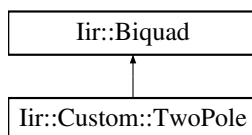
The documentation for this class was generated from the following file:

- 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

| | |
|------------------|--------------------|
| <i>poleRho</i> | Radius of the pole |
| <i>poleTheta</i> | Angle of the pole |
| <i>zeroRho</i> | Radius of the zero |
| <i>zeroTheta</i> | 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 `lir.h`

```

00001
00035 #ifndef IIR_H
00036 #define IIR_H
00037
00038 //
00039 // Include this file in your application to get everything
00040 //
00041
00042 #include "iir/Common.h"
00043
00044 #include "iir/Biquad.h"
00045 #include "iir/Cascade.h"
00046 #include "iir/PoleFilter.h"
00047 #include "iir/State.h"
  
```

```

00048
00049 #include "iir/Butterworth.h"
00050 #include "iir/ChebyshevI.h"
00051 #include "iir/ChebyshevII.h"
00052 #include "iir/Custom.h"
00053 #include "iir/RBJ.h"
00054
00055 #endif

```

8.2 Biquad.h

```

00001
00036 #ifndef IIR1_BIQUAD_H
00037 #define IIR1_BIQUAD_H
00038
00039 #include "Common.h"
00040 #include "MathSupplement.h"
00041 #include "Types.h"
00042
00043 namespace Iir {
00044
00045     struct IIR_EXPORT BiquadPoleState;
00046
00047     /*
00048     * Holds coefficients for a second order Infinite Impulse Response
00049     * digital filter. This is the building block for all IIR filters.
00050     */
00051     class IIR_EXPORT Biquad {
00052     public:
00053
00054         Biquad() = default;
00055
00056         complex_t response (double normalizedFrequency) const;
00061
00062         std::vector<PoleZeroPair> getPoleZeros () const;
00066
00067         double getA0 () const { return m_a0; }
00071
00072         double getA1 () const { return m_a1*m_a0; }
00076
00077         double getA2 () const { return m_a2*m_a0; }
00081
00082         double getB0 () const { return m_b0*m_a0; }
00086
00087         double getB1 () const { return m_b1*m_a0; }
00091
00092         double getB2 () const { return m_b2*m_a0; }
00096
00097         template <class StateType>
00104         inline double filter(double s, StateType& state) const
00105         {
00106             return state.filter(s, *this);
00107         }
00108
00109     public:
00110         void setCoefficients (double a0, double a1, double a2,
00120                             double b0, double b1, double b2);
00121
00122         void setOnePole (complex_t pole, complex_t zero);
00126
00127         void setTwoPole (complex_t pole1, complex_t zero1,
00131                             complex_t pole2, complex_t zero2);
00132
00133         void setPoleZeroPair (const PoleZeroPair& pair)
00137         {
00138             if (pair.isSinglePole ())
00139                 setOnePole (pair.poles.first, pair.zeros.first);
00140             else
00141                 setTwoPole (pair.poles.first, pair.zeros.first,
00142                             pair.poles.second, pair.zeros.second);
00143         }
00144
00145         void setPoleZeroForm (const BiquadPoleState& bps);
00146
00147         void setIdentity ();
00151
00152         void applyScale (double scale);
00153
00154     public:
00155         double m_a0 = 1.0;
00160         double m_a1 = 0.0;
00161         double m_a2 = 0.0;
00162         double m_b1 = 0.0;
00163         double m_b2 = 0.0;
00164         double m_b0 = 1.0;
00165

```

```

00166     };
00167
00168 //-----
00169
00170
00175     struct IIR_EXPORT BiquadPoleState : PoleZeroPair
00176     {
00177         BiquadPoleState () = default;
00178
00179         explicit BiquadPoleState (const Biquad& s);
00180
00181         double gain = 1.0;
00182     };
00183
00184 }
00185
00186 #endif

```

8.3 Butterworth.h

```

00001
00036 #ifndef IIR1_BUTTERWORTH_H
00037 #define IIR1_BUTTERWORTH_H
00038
00039 #include "Common.h"
00040 #include "Cascade.h"
00041 #include "PoleFilter.h"
00042 #include "State.h"
00043
00044 namespace Iir {
00045
00052 namespace Butterworth {
00053
00057 class IIR_EXPORT AnalogLowPass : public LayoutBase
00058 {
00059 public:
00060     AnalogLowPass ();
00061
00062     void design (const int numPoles);
00063
00064 private:
00065     int m_numPoles = 0;
00066 };
00067
00068 //-----
00069
00073 class IIR_EXPORT AnalogLowShelf : public LayoutBase
00074 {
00075 public:
00076     AnalogLowShelf ();
00077
00078     void design (int numPoles, double gainDb);
00079
00080 private:
00081     int m_numPoles = 0;
00082     double m_gainDb = 0.0;
00083 };
00084
00085 //-----
00086
00087 struct IIR_EXPORT LowPassBase : PoleFilterBase <AnalogLowPass>
00088 {
00089     void setup (int order,
00090                 double cutoffFrequency);
00091 };
00092
00093 struct IIR_EXPORT HighPassBase : PoleFilterBase <AnalogLowPass>
00094 {
00095     void setup (int order,
00096                 double cutoffFrequency);
00097 };
00098
00099 struct IIR_EXPORT BandPassBase : PoleFilterBase <AnalogLowPass>
00100 {
00101     void setup (int order,
00102                 double centerFrequency,
00103                 double widthFrequency);
00104 };
00105
00106 struct IIR_EXPORT BandStopBase : PoleFilterBase <AnalogLowPass>
00107 {
00108     void setup (int order,
00109                 double centerFrequency,
00110                 double widthFrequency);
00111 };
00112

```

```

00113 struct IIR_EXPORT LowShelfBase : PoleFilterBase <AnalogLowShelf>
00114 {
00115     void setup (int order,
00116                 double cutoffFrequency,
00117                 double gainDb);
00118 };
00119
00120 struct IIR_EXPORT HighShelfBase : PoleFilterBase <AnalogLowShelf>
00121 {
00122     void setup (int order,
00123                 double cutoffFrequency,
00124                 double gainDb);
00125 };
00126
00127 struct IIR_EXPORT BandShelfBase : PoleFilterBase <AnalogLowShelf>
00128 {
00129     void setup (int order,
00130                 double centerFrequency,
00131                 double widthFrequency,
00132                 double gainDb);
00133 };
00134
00135 //-----
00136
00137 //
00138 // Filters for the user
00139 //
00140
00146 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00147 struct LowPass : PoleFilter <LowPassBase, StateType, FilterOrder>
00148 {
00154     void setup (double sampleRate,
00155                 double cutoffFrequency) {
00156         LowPassBase::setup (FilterOrder,
00157                             cutoffFrequency / sampleRate);
00158     }
00159
00166     void setup (int reqOrder,
00167                 double sampleRate,
00168                 double cutoffFrequency) {
00169         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00170         LowPassBase::setup (reqOrder,
00171                             cutoffFrequency / sampleRate);
00172     }
00173
00174
00179     void setupN(double cutoffFrequency) {
00180         LowPassBase::setup (FilterOrder,
00181                             cutoffFrequency);
00182     }
00183
00189     void setupN(int reqOrder,
00190                 double cutoffFrequency) {
00191         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00192         LowPassBase::setup (reqOrder,
00193                             cutoffFrequency);
00194     }
00195 };
00196
00202 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00203 struct HighPass : PoleFilter <HighPassBase, StateType, FilterOrder>
00204 {
00205
00211     void setup (double sampleRate,
00212                 double cutoffFrequency) {
00213         HighPassBase::setup (FilterOrder,
00214                             cutoffFrequency / sampleRate);
00215     }
00222     void setup (int reqOrder,
00223                 double sampleRate,
00224                 double cutoffFrequency) {
00225         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00226         HighPassBase::setup (reqOrder,
00227                             cutoffFrequency / sampleRate);
00228     }
00229
00230
00235     void setupN(double cutoffFrequency) {
00236         HighPassBase::setup (FilterOrder,
00237                             cutoffFrequency);
00238     }
00244     void setupN(int reqOrder,
00245                 double cutoffFrequency) {
00246         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00247         HighPassBase::setup (reqOrder,
00248                             cutoffFrequency);
00249     }

```

```

00250 };
00251
00257 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00258 struct BandPass : PoleFilter <BandPassBase, StateType, FilterOrder, FilterOrder*2>
00259 {
00266     void setup (double sampleRate,
00267                 double centerFrequency,
00268                 double widthFrequency) {
00269         BandPassBase::setup(FilterOrder,
00270                               centerFrequency / sampleRate,
00271                               widthFrequency / sampleRate);
00272     }
00273
00281     void setup (int reqOrder,
00282                 double sampleRate,
00283                 double centerFrequency,
00284                 double widthFrequency) {
00285         if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
00286         BandPassBase::setup(reqOrder,
00287                               centerFrequency / sampleRate,
00288                               widthFrequency / sampleRate);
00289     }
00290
00291
00292
00298     void setupN(double centerFrequency,
00299                 double widthFrequency) {
00300         BandPassBase::setup(FilterOrder,
00301                               centerFrequency,
00302                               widthFrequency);
00303     }
00304
00311     void setupN(int reqOrder,
00312                 double centerFrequency,
00313                 double widthFrequency) {
00314         if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
00315         BandPassBase::setup(reqOrder,
00316                               centerFrequency,
00317                               widthFrequency);
00318     }
00319 };
00320
00321
00327 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00328 struct BandStop : PoleFilter <BandStopBase, StateType, FilterOrder, FilterOrder*2>
00329 {
00336     void setup (double sampleRate,
00337                 double centerFrequency,
00338                 double widthFrequency) {
00339         BandStopBase::setup (FilterOrder,
00340                               centerFrequency / sampleRate,
00341                               widthFrequency / sampleRate);
00342     }
00343
00351     void setup (int reqOrder,
00352                 double sampleRate,
00353                 double centerFrequency,
00354                 double widthFrequency) {
00355         if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
00356         BandStopBase::setup (reqOrder,
00357                               centerFrequency / sampleRate,
00358                               widthFrequency / sampleRate);
00359     }
00360
00361
00362
00368     void setupN(double centerFrequency,
00369                 double widthFrequency) {
00370         BandStopBase::setup (FilterOrder,
00371                               centerFrequency,
00372                               widthFrequency);
00373     }
00374
00381     void setupN(int reqOrder,
00382                 double centerFrequency,
00383                 double widthFrequency) {
00384         if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
00385         BandStopBase::setup (reqOrder,
00386                               centerFrequency,
00387                               widthFrequency);
00388     }
00389
00390 };
00391
00398 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00399 struct LowShelf : PoleFilter <LowShelfBase, StateType, FilterOrder>
00400 {

```

```

00407     void setup (double sampleRate,
00408                 double cutoffFrequency,
00409                 double gainDb) {
00410         LowShelfBase::setup (FilterOrder,
00411                             cutoffFrequency / sampleRate,
00412                             gainDb);
00413     }
00414
00422     void setup (int reqOrder,
00423                 double sampleRate,
00424                 double cutoffFrequency,
00425                 double gainDb) {
00426         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00427         LowShelfBase::setup (reqOrder,
00428                             cutoffFrequency / sampleRate,
00429                             gainDb);
00430     }
00431
00432
00433
00434
00440     void setupN(double cutoffFrequency,
00441                 double gainDb) {
00442         LowShelfBase::setup (FilterOrder,
00443                             cutoffFrequency,
00444                             gainDb);
00445     }
00446
00453     void setupN(int reqOrder,
00454                 double cutoffFrequency,
00455                 double gainDb) {
00456         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00457         LowShelfBase::setup (reqOrder,
00458                             cutoffFrequency,
00459                             gainDb);
00460     }
00461
00462 };
00463
00464
00471 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00472 struct HighShelf : PoleFilter <HighShelfBase, StateType, FilterOrder>
00473 {
00480     void setup (double sampleRate,
00481                 double cutoffFrequency,
00482                 double gainDb) {
00483         HighShelfBase::setup (FilterOrder,
00484                               cutoffFrequency / sampleRate,
00485                               gainDb);
00486     }
00487
00495     void setup (int reqOrder,
00496                 double sampleRate,
00497                 double cutoffFrequency,
00498                 double gainDb) {
00499         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00500         HighShelfBase::setup (reqOrder,
00501                               cutoffFrequency / sampleRate,
00502                               gainDb);
00503     }
00504
00505
00506
00512     void setupN(double cutoffFrequency,
00513                 double gainDb) {
00514         HighShelfBase::setup (FilterOrder,
00515                               cutoffFrequency,
00516                               gainDb);
00517     }
00518
00525     void setupN(int reqOrder,
00526                 double cutoffFrequency,
00527                 double gainDb) {
00528         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00529         HighShelfBase::setup (reqOrder,
00530                               cutoffFrequency,
00531                               gainDb);
00532     }
00533 };
00534
00535
00542 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00543 struct BandShelf : PoleFilter <BandShelfBase, StateType, FilterOrder, FilterOrder*2>
00544 {
00552     void setup (double sampleRate,
00553                 double centerFrequency,
00554                 double widthFrequency,

```

```

00555         double gainDb) {
00556             BandShelfBase::setup (FilterOrder,
00557                                   centerFrequency / sampleRate,
00558                                   widthFrequency / sampleRate,
00559                                   gainDb);
00560     }
00561
00570     void setup (int reqOrder,
00571                 double sampleRate,
00572                 double centerFrequency,
00573                 double widthFrequency,
00574                 double gainDb) {
00575         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00576         BandShelfBase::setup (reqOrder,
00577                               centerFrequency / sampleRate,
00578                               widthFrequency / sampleRate,
00579                               gainDb);
00580     }
00581
00582
00583
00590     void setupN(double centerFrequency,
00591                 double widthFrequency,
00592                 double gainDb) {
00593         BandShelfBase::setup (FilterOrder,
00594                               centerFrequency,
00595                               widthFrequency,
00596                               gainDb);
00597     }
00598
00606     void setupN(int reqOrder,
00607                 double centerFrequency,
00608                 double widthFrequency,
00609                 double gainDb) {
00610         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00611         BandShelfBase::setup (reqOrder,
00612                               centerFrequency,
00613                               widthFrequency,
00614                               gainDb);
00615     }
00616 };
00617
00618 }
00619
00620 }
00621
00622 #endif
00623

```

8.4 Cascade.h

```

00001
00036 #ifndef IIR1_CASCADE_H
00037 #define IIR1_CASCADE_H
00038
00039 #include "Common.h"
00040 #include "Biquad.h"
00041 #include "Layout.h"
00042 #include "MathSupplement.h"
00043
00044 namespace Iir {
00045
00049     class IIR_EXPORT Cascade
00050     {
00051     public:
00052
00053         Cascade () = default;
00054
00059         struct IIR_EXPORT Storage
00060         {
00061             Storage() = delete;
00062             Storage(const int maxNumBiquads, Biquad* const biquadArray) : maxStages(maxNumBiquads),
stageArray(biquadArray) {}
00063             const int maxStages = 0;
00064             Biquad* const stageArray = nullptr;
00065         };
00066
00070         int getNumStages () const
00071         {
00072             return m_numStages;
00073         }
00074
00078         const Biquad& operator[] (int index)
00079         {
00080             if ((index < 0) || (index >= m_numStages))
00081                 throw_invalid_argument("Index out of bounds.");

```

```

00082         return m_stageArray[index];
00083     }
00084
00089     complex_t response (double normalizedFrequency) const;
00090
00094     std::vector<PoleZeroPair> getPoleZeros () const;
00095
00096     void setCascadeStorage (const Storage& storage);
00097
00098     void applyScale (double scale);
00099
00100     void setLayout (const LayoutBase& proto);
00101
00102     private:
00103     int m_numStages = 0;
00104     int m_maxStages = 0;
00105     Biquad* m_stageArray = nullptr;
00106     };
00107
00108
00109 //-----
00110
00115     template <int MaxStages, class StateType>
00116     class CascadeStages {
00117
00118     public:
00119     CascadeStages() = default;
00120
00121
00122     public:
00126     void reset ()
00127     {
00128         for (auto &state: m_states)
00129             state.reset();
00130     }
00131
00132     public:
00138     void setup (const double (&sosCoefficients)[MaxStages][6]) {
00139         for (int i = 0; i < MaxStages; i++) {
00140             m_stages[i].setCoefficients(
00141                 sosCoefficients[i][3],
00142                 sosCoefficients[i][4],
00143                 sosCoefficients[i][5],
00144                 sosCoefficients[i][0],
00145                 sosCoefficients[i][1],
00146                 sosCoefficients[i][2]);
00147         }
00148     }
00149
00150     public:
00156     template <typename Sample>
00157     inline Sample filter(const Sample in)
00158     {
00159         double out = in;
00160         StateType* state = m_states;
00161         for (const auto &stage: m_stages)
00162             out = (state++)->filter(out, stage);
00163         return static_cast<Sample> (out);
00164     }
00165
00169     const Cascade::Storage getCascadeStorage()
00170     {
00171         const Cascade::Storage s(MaxStages, m_stages);
00172         return s;
00173     }
00174
00175     private:
00176     Biquad m_stages[MaxStages] = {};
00177     StateType m_states[MaxStages] = {};
00178     };
00179
00180 }
00181
00182 #endif

```

8.5 ChebyshevI.h

```

00001
00036 #ifndef IIR1_CHEBYSHEVI_H
00037 #define IIR1_CHEBYSHEVI_H
00038
00039 #include "Common.h"
00040 #include "Cascade.h"
00041 #include "PoleFilter.h"
00042 #include "State.h"
00043

```

```

00044 namespace Iir {
00045
00050 namespace ChebyshevI {
00051
00055 class IIR_EXPORT AnalogLowPass : public LayoutBase
00056 {
00057 public:
00058     AnalogLowPass ();
00059
00060     void design (const int numPoles,
00061                 double rippleDb);
00062
00063 private:
00064     int m_numPoles = 0;
00065     double m_rippleDb = 0.0;
00066 };
00067
00071 class IIR_EXPORT AnalogLowShelf : public LayoutBase
00072 {
00073 public:
00074     AnalogLowShelf ();
00075
00076     void design (int numPoles,
00077                 double gainDb,
00078                 double rippleDb);
00079
00080 private:
00081     int m_numPoles = 0;
00082     double m_rippleDb = 0.0;
00083     double m_gainDb = 0.0;
00084 };
00085
00086 //-----
00087
00088 struct IIR_EXPORT LowPassBase : PoleFilterBase <AnalogLowPass>
00089 {
00090     void setup (int order,
00091                 double cutoffFrequency,
00092                 double rippleDb);
00093 };
00094
00095 struct IIR_EXPORT HighPassBase : PoleFilterBase <AnalogLowPass>
00096 {
00097     void setup (int order,
00098                 double cutoffFrequency,
00099                 double rippleDb);
00100 };
00101
00102 struct IIR_EXPORT BandPassBase : PoleFilterBase <AnalogLowPass>
00103 {
00104     void setup (int order,
00105                 double centerFrequency,
00106                 double widthFrequency,
00107                 double rippleDb);
00108 };
00109
00110 struct IIR_EXPORT BandStopBase : PoleFilterBase <AnalogLowPass>
00111 {
00112     void setup (int order,
00113                 double centerFrequency,
00114                 double widthFrequency,
00115                 double rippleDb);
00116 };
00117
00118 struct IIR_EXPORT LowShelfBase : PoleFilterBase <AnalogLowShelf>
00119 {
00120     void setup (int order,
00121                 double cutoffFrequency,
00122                 double gainDb,
00123                 double rippleDb);
00124 };
00125
00126 struct IIR_EXPORT HighShelfBase : PoleFilterBase <AnalogLowShelf>
00127 {
00128     void setup (int order,
00129                 double cutoffFrequency,
00130                 double gainDb,
00131                 double rippleDb);
00132 };
00133
00134 struct IIR_EXPORT BandShelfBase : PoleFilterBase <AnalogLowShelf>
00135 {
00136     void setup (int order,
00137                 double centerFrequency,
00138                 double widthFrequency,
00139                 double gainDb,
00140                 double rippleDb);

```

```

00141 };
00142
00143 //-----
00144 //
00145 //
00146 // Userland filters
00147 //
00148
00154 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00155     struct LowPass : PoleFilter <LowPassBase, StateType, FilterOrder>
00156     {
00163         void setup (double sampleRate,
00164                     double cutoffFrequency,
00165                     double rippleDb) {
00166             LowPassBase::setup (FilterOrder,
00167                                 cutoffFrequency / sampleRate,
00168                                 rippleDb);
00169         }
00170
00178         void setup (int reqOrder,
00179                     double sampleRate,
00180                     double cutoffFrequency,
00181                     double rippleDb) {
00182             if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00183             LowPassBase::setup (reqOrder,
00184                                 cutoffFrequency / sampleRate,
00185                                 rippleDb);
00186         }
00187
00188
00189
00195         void setupN(double cutoffFrequency,
00196                     double rippleDb) {
00197             LowPassBase::setup (FilterOrder,
00198                                 cutoffFrequency,
00199                                 rippleDb);
00200         }
00201
00208         void setupN(int reqOrder,
00209                     double cutoffFrequency,
00210                     double rippleDb) {
00211             if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00212             LowPassBase::setup (reqOrder,
00213                                 cutoffFrequency,
00214                                 rippleDb);
00215         }
00216 };
00217
00223 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00224     struct HighPass : PoleFilter <HighPassBase, StateType, FilterOrder>
00225     {
00232         void setup (double sampleRate,
00233                     double cutoffFrequency,
00234                     double rippleDb) {
00235             HighPassBase::setup (FilterOrder,
00236                                 cutoffFrequency / sampleRate,
00237                                 rippleDb);
00238         }
00239
00247         void setup (int reqOrder,
00248                     double sampleRate,
00249                     double cutoffFrequency,
00250                     double rippleDb) {
00251             if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00252             HighPassBase::setup (reqOrder,
00253                                 cutoffFrequency / sampleRate,
00254                                 rippleDb);
00255         }
00256
00257
00258
00264         void setupN(double cutoffFrequency,
00265                     double rippleDb) {
00266             HighPassBase::setup (FilterOrder,
00267                                 cutoffFrequency,
00268                                 rippleDb);
00269         }
00270
00277         void setupN(int reqOrder,
00278                     double cutoffFrequency,
00279                     double rippleDb) {
00280             if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00281             HighPassBase::setup (reqOrder,
00282                                 cutoffFrequency,
00283                                 rippleDb);
00284         }
00285 };

```

```

00286
00292 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00293     struct BandPass : PoleFilter <BandPassBase, StateType, FilterOrder, FilterOrder*2>
00294     {
00302         void setup (double sampleRate,
00303                     double centerFrequency,
00304                     double widthFrequency,
00305                     double rippleDb) {
00306             BandPassBase::setup (FilterOrder,
00307                                   centerFrequency / sampleRate,
00308                                   widthFrequency / sampleRate,
00309                                   rippleDb);
00310         }
00311
00320         void setup (int reqOrder,
00321                     double sampleRate,
00322                     double centerFrequency,
00323                     double widthFrequency,
00324                     double rippleDb) {
00325             if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00326             BandPassBase::setup (reqOrder,
00327                                   centerFrequency / sampleRate,
00328                                   widthFrequency / sampleRate,
00329                                   rippleDb);
00330         }
00331
00332
00333
00340         void setupN(double centerFrequency,
00341                     double widthFrequency,
00342                     double rippleDb) {
00343             BandPassBase::setup (FilterOrder,
00344                                   centerFrequency,
00345                                   widthFrequency,
00346                                   rippleDb);
00347         }
00348
00356         void setupN(int reqOrder,
00357                     double centerFrequency,
00358                     double widthFrequency,
00359                     double rippleDb) {
00360             if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00361             BandPassBase::setup (reqOrder,
00362                                   centerFrequency,
00363                                   widthFrequency,
00364                                   rippleDb);
00365         }
00366     };
00367
00373 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00374     struct BandStop : PoleFilter <BandStopBase, StateType, FilterOrder, FilterOrder*2>
00375     {
00383         void setup (double sampleRate,
00384                     double centerFrequency,
00385                     double widthFrequency,
00386                     double rippleDb) {
00387             BandStopBase::setup (FilterOrder,
00388                                   centerFrequency / sampleRate,
00389                                   widthFrequency / sampleRate,
00390                                   rippleDb);
00391         }
00392
00401         void setup (int reqOrder,
00402                     double sampleRate,
00403                     double centerFrequency,
00404                     double widthFrequency,
00405                     double rippleDb) {
00406             if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00407             BandStopBase::setup (reqOrder,
00408                                   centerFrequency / sampleRate,
00409                                   widthFrequency / sampleRate,
00410                                   rippleDb);
00411         }
00412
00413
00414
00421         void setupN(double centerFrequency,
00422                     double widthFrequency,
00423                     double rippleDb) {
00424             BandStopBase::setup (FilterOrder,
00425                                   centerFrequency,
00426                                   widthFrequency,
00427                                   rippleDb);
00428         }
00429
00437         void setupN(int reqOrder,
00438                     double centerFrequency,

```

```

00439         double widthFrequency,
00440         double rippleDb) {
00441     if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00442     BandStopBase::setup (reqOrder,
00443         centerFrequency,
00444         widthFrequency,
00445         rippleDb);
00446 }
00447
00448 };
00449
00450 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00451 struct LowShelf : PoleFilter <LowShelfBase, StateType, FilterOrder>
00452 {
00453     void setup (double sampleRate,
00454         double cutoffFrequency,
00455         double gainDb,
00456         double rippleDb) {
00457         LowShelfBase::setup (FilterOrder,
00458             cutoffFrequency / sampleRate,
00459             gainDb,
00460             rippleDb);
00461     }
00462
00463     void setup (int reqOrder,
00464         double sampleRate,
00465         double cutoffFrequency,
00466         double gainDb,
00467         double rippleDb) {
00468         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00469         LowShelfBase::setup (reqOrder,
00470             cutoffFrequency / sampleRate,
00471             gainDb,
00472             rippleDb);
00473     }
00474
00475     void setupN(double cutoffFrequency,
00476         double gainDb,
00477         double rippleDb) {
00478         LowShelfBase::setup (FilterOrder,
00479             cutoffFrequency,
00480             gainDb,
00481             rippleDb);
00482     }
00483
00484     void setupN(int reqOrder,
00485         double cutoffFrequency,
00486         double gainDb,
00487         double rippleDb) {
00488         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00489         LowShelfBase::setup (reqOrder,
00490             cutoffFrequency,
00491             gainDb,
00492             rippleDb);
00493     }
00494 };
00495
00496 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00497 struct HighShelf : PoleFilter <HighShelfBase, StateType, FilterOrder>
00498 {
00499     void setup (double sampleRate,
00500         double cutoffFrequency,
00501         double gainDb,
00502         double rippleDb) {
00503         HighShelfBase::setup (FilterOrder,
00504             cutoffFrequency / sampleRate,
00505             gainDb,
00506             rippleDb);
00507     }
00508
00509     void setup (int reqOrder,
00510         double sampleRate,
00511         double cutoffFrequency,
00512         double gainDb,
00513         double rippleDb) {
00514         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00515         HighShelfBase::setup (reqOrder,
00516             cutoffFrequency / sampleRate,
00517             gainDb,
00518             rippleDb);
00519     }
00520 };
00521
00522 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00523 struct LowPass : PoleFilter <LowPassBase, StateType, FilterOrder>
00524 {
00525     void setup (double sampleRate,
00526         double cutoffFrequency,
00527         double gainDb,
00528         double rippleDb) {
00529         LowPassBase::setup (FilterOrder,
00530             cutoffFrequency / sampleRate,
00531             gainDb,
00532             rippleDb);
00533     }
00534
00535     void setup (int reqOrder,
00536         double sampleRate,
00537         double cutoffFrequency,
00538         double gainDb,
00539         double rippleDb) {
00540         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00541         LowPassBase::setup (reqOrder,
00542             cutoffFrequency / sampleRate,
00543             gainDb,
00544             rippleDb);
00545     }
00546 };
00547
00548 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00549 struct HighPass : PoleFilter <HighPassBase, StateType, FilterOrder>
00550 {
00551     void setup (double sampleRate,
00552         double cutoffFrequency,
00553         double gainDb,
00554         double rippleDb) {
00555         HighPassBase::setup (FilterOrder,
00556             cutoffFrequency / sampleRate,
00557             gainDb,
00558             rippleDb);
00559     }
00560
00561     void setup (int reqOrder,
00562         double sampleRate,
00563         double cutoffFrequency,
00564         double gainDb,
00565         double rippleDb) {
00566         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00567         HighPassBase::setup (reqOrder,
00568             cutoffFrequency / sampleRate,
00569             gainDb,
00570             rippleDb);
00571     }
00572 };
00573
00574 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00575 struct BandPass : PoleFilter <BandPassBase, StateType, FilterOrder>
00576 {
00577     void setup (double sampleRate,
00578         double centerFrequency,
00579         double widthFrequency,
00580         double gainDb,
00581         double rippleDb) {
00582         BandPassBase::setup (FilterOrder,
00583             centerFrequency / sampleRate,
00584             widthFrequency / sampleRate,
00585             gainDb,
00586             rippleDb);
00587     }
00588
00589     void setup (int reqOrder,
00590         double sampleRate,
00591         double centerFrequency,
00592         double widthFrequency,
00593         double gainDb,
00594         double rippleDb) {
00595         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00596         BandPassBase::setup (reqOrder,
00597             centerFrequency / sampleRate,
00598             widthFrequency / sampleRate,
00599             gainDb,
00600             rippleDb);
00601     }
00602 };

```

```

00585         void setupN(double cutoffFrequency,
00586                     double gainDb,
00587                     double rippleDb) {
00588             HighShelfBase::setup (FilterOrder,
00589                                   cutoffFrequency,
00590                                   gainDb,
00591                                   rippleDb);
00592         }
00593
00601         void setupN(int reqOrder,
00602                     double cutoffFrequency,
00603                     double gainDb,
00604                     double rippleDb) {
00605             if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00606             HighShelfBase::setup (reqOrder,
00607                                   cutoffFrequency,
00608                                   gainDb,
00609                                   rippleDb);
00610         }
00611     };
00612 };
00613
00619 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00620 struct BandShelf : PoleFilter <BandShelfBase, StateType, FilterOrder, FilterOrder*2>
00621 {
00630     void setup (double sampleRate,
00631                 double centerFrequency,
00632                 double widthFrequency,
00633                 double gainDb,
00634                 double rippleDb) {
00635         BandShelfBase::setup (FilterOrder,
00636                               centerFrequency / sampleRate,
00637                               widthFrequency / sampleRate,
00638                               gainDb,
00639                               rippleDb);
00640     }
00641
00642     void setup (int reqOrder,
00653                 double sampleRate,
00654                 double centerFrequency,
00655                 double widthFrequency,
00656                 double gainDb,
00657                 double rippleDb) {
00658         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00659         BandShelfBase::setup (reqOrder,
00660                               centerFrequency / sampleRate,
00661                               widthFrequency / sampleRate,
00662                               gainDb,
00663                               rippleDb);
00664     }
00665
00666     void setupN(double centerFrequency,
00678                 double widthFrequency,
00679                 double gainDb,
00680                 double rippleDb) {
00681         BandShelfBase::setup (FilterOrder,
00682                               centerFrequency,
00683                               widthFrequency,
00684                               gainDb,
00685                               rippleDb);
00686     }
00687
00688     void setupN(int reqOrder,
00698                 double centerFrequency,
00699                 double widthFrequency,
00700                 double gainDb,
00701                 double rippleDb) {
00702         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00703         BandShelfBase::setup (reqOrder,
00704                               centerFrequency,
00705                               widthFrequency,
00706                               gainDb,
00707                               rippleDb);
00708     }
00709 };
00710
00711 };
00712
00713 }
00714
00715 }

```

```
00716
00717 #endif
```

8.6 ChebyshevII.h

```
00001
00036 #ifndef IIR1_CHEBYSHEVII_H
00037 #define IIR1_CHEBYSHEVII_H
00038
00039 #include "Common.h"
00040 #include "Cascade.h"
00041 #include "PoleFilter.h"
00042 #include "State.h"
00043
00044 namespace Iir {
00045
00053 namespace ChebyshevII {
00054
00058 class IIR_EXPORT AnalogLowPass : public LayoutBase
00059 {
00060 public:
00061     AnalogLowPass ();
00062
00063     void design (const int numPoles,
00064                 double stopBandDb);
00065
00066 private:
00067     int m_numPoles = 0;
00068     double m_stopBandDb = 0.0;
00069 };
00070
00071
00075 class IIR_EXPORT AnalogLowShelf : public LayoutBase
00076 {
00077 public:
00078     AnalogLowShelf ();
00079
00080     void design (int numPoles,
00081                 double gainDb,
00082                 double stopBandDb);
00083
00084 private:
00085     int m_numPoles = 0;
00086     double m_stopBandDb = 0.0;
00087     double m_gainDb = 0.0;
00088 };
00089
00090 //-----
00091
00092 struct IIR_EXPORT LowPassBase : PoleFilterBase <AnalogLowPass>
00093 {
00094     void setup (int order,
00095                 double cutoffFrequency,
00096                 double stopBandDb);
00097 };
00098
00099 struct IIR_EXPORT HighPassBase : PoleFilterBase <AnalogLowPass>
00100 {
00101     void setup (int order,
00102                 double cutoffFrequency,
00103                 double stopBandDb);
00104 };
00105
00106 struct IIR_EXPORT BandPassBase : PoleFilterBase <AnalogLowPass>
00107 {
00108     void setup (int order,
00109                 double centerFrequency,
00110                 double widthFrequency,
00111                 double stopBandDb);
00112 };
00113
00114 struct IIR_EXPORT BandStopBase : PoleFilterBase <AnalogLowPass>
00115 {
00116     void setup (int order,
00117                 double centerFrequency,
00118                 double widthFrequency,
00119                 double stopBandDb);
00120 };
00121
00122 struct IIR_EXPORT LowShelfBase : PoleFilterBase <AnalogLowShelf>
00123 {
00124     void setup (int order,
00125                 double cutoffFrequency,
00126                 double gainDb,
00127                 double stopBandDb);
00128 };
```

```

00129
00130 struct IIR_EXPORT HighShelfBase : PoleFilterBase <AnalogLowShelf>
00131 {
00132     void setup (int order,
00133                 double cutoffFrequency,
00134                 double gainDb,
00135                 double stopBandDb);
00136 };
00137
00138 struct IIR_EXPORT BandShelfBase : PoleFilterBase <AnalogLowShelf>
00139 {
00140     void setup (int order,
00141                 double centerFrequency,
00142                 double widthFrequency,
00143                 double gainDb,
00144                 double stopBandDb);
00145 };
00146
00147 //-----
00148 //
00149 // Userland filters
00150 //
00151 //
00152
00153 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00154 struct LowPass : PoleFilter <LowPassBase, StateType, FilterOrder>
00155 {
00156     void setup (double sampleRate,
00157                 double cutoffFrequency,
00158                 double stopBandDb) {
00159         LowPassBase::setup (FilterOrder,
00160                             cutoffFrequency / sampleRate,
00161                             stopBandDb);
00162     }
00163
00164     void setup (int reqOrder,
00165                 double sampleRate,
00166                 double cutoffFrequency,
00167                 double stopBandDb) {
00168         if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
00169         LowPassBase::setup (reqOrder,
00170                             cutoffFrequency / sampleRate,
00171                             stopBandDb);
00172     }
00173
00174     void setupN(double cutoffFrequency,
00175                 double stopBandDb) {
00176         LowPassBase::setup (FilterOrder,
00177                             cutoffFrequency,
00178                             stopBandDb);
00179     }
00180
00181     void setupN(int reqOrder,
00182                 double cutoffFrequency,
00183                 double stopBandDb) {
00184         if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
00185         LowPassBase::setup (reqOrder,
00186                             cutoffFrequency,
00187                             stopBandDb);
00188     }
00189 };
00190
00191
00192
00193
00194
00195
00201
00202
00203
00204
00205
00206
00207
00214
00215
00216
00217
00218
00219
00220
00221
00222
00223 };
00224
00230 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00231 struct HighPass : PoleFilter <HighPassBase, StateType, FilterOrder>
00232 {
00233     void setup (double sampleRate,
00234                 double cutoffFrequency,
00235                 double stopBandDb) {
00236         HighPassBase::setup (FilterOrder,
00237                             cutoffFrequency / sampleRate,
00238                             stopBandDb);
00239     }
00240
00241     void setup (int reqOrder,
00242                 double sampleRate,
00243                 double cutoffFrequency,
00244                 double stopBandDb) {
00245         if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
00246         HighPassBase::setup (reqOrder,
00247                             cutoffFrequency / sampleRate,
00248                             stopBandDb);
00249     }
00250 };
00251
00252
00253
00254
00255
00256
00257
00258
00259
00260
00261
00262

```

```

00263
00264
00265
00266
00272 void setupN(double cutoffFrequency,
00273             double stopBandDb) {
00274     HighPassBase::setup (FilterOrder,
00275                          cutoffFrequency,
00276                          stopBandDb);
00277 }
00278
00285 void setupN(int reqOrder,
00286             double cutoffFrequency,
00287             double stopBandDb) {
00288     if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00289     HighPassBase::setup (reqOrder,
00290                          cutoffFrequency,
00291                          stopBandDb);
00292 }
00293
00294 };
00295
00301 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00302 struct BandPass : PoleFilter <BandPassBase, StateType, FilterOrder, FilterOrder*2>
00303 {
00311     void setup (double sampleRate,
00312                double centerFrequency,
00313                double widthFrequency,
00314                double stopBandDb) {
00315         BandPassBase::setup (FilterOrder,
00316                              centerFrequency / sampleRate,
00317                              widthFrequency / sampleRate,
00318                              stopBandDb);
00319     }
00320
00329 void setup (int reqOrder,
00330             double sampleRate,
00331             double centerFrequency,
00332             double widthFrequency,
00333             double stopBandDb) {
00334     if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00335     BandPassBase::setup (reqOrder,
00336                          centerFrequency / sampleRate,
00337                          widthFrequency / sampleRate,
00338                          stopBandDb);
00339 }
00340
00341
00342
00343
00350 void setupN(double centerFrequency,
00351            double widthFrequency,
00352            double stopBandDb) {
00353     BandPassBase::setup (FilterOrder,
00354                          centerFrequency,
00355                          widthFrequency,
00356                          stopBandDb);
00357 }
00358
00366 void setupN(int reqOrder,
00367            double centerFrequency,
00368            double widthFrequency,
00369            double stopBandDb) {
00370     if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00371     BandPassBase::setup (reqOrder,
00372                          centerFrequency,
00373                          widthFrequency,
00374                          stopBandDb);
00375 }
00376 };
00377
00383 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00384 struct BandStop : PoleFilter <BandStopBase, StateType, FilterOrder, FilterOrder*2>
00385 {
00393     void setup (double sampleRate,
00394                double centerFrequency,
00395                double widthFrequency,
00396                double stopBandDb) {
00397         BandStopBase::setup (FilterOrder,
00398                              centerFrequency / sampleRate,
00399                              widthFrequency / sampleRate,
00400                              stopBandDb);
00401     }
00402
00411 void setup (int reqOrder,
00412            double sampleRate,
00413            double centerFrequency,

```

```

00414         double widthFrequency,
00415         double stopBandDb) {
00416     if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
00417     BandStopBase::setup (reqOrder,
00418         centerFrequency / sampleRate,
00419         widthFrequency / sampleRate,
00420         stopBandDb);
00421 }
00422
00423
00424
00425
00432 void setupN(double centerFrequency,
00433     double widthFrequency,
00434     double stopBandDb) {
00435     BandStopBase::setup (FilterOrder,
00436         centerFrequency,
00437         widthFrequency,
00438         stopBandDb);
00439 }
00440
00448 void setupN(int reqOrder,
00449     double centerFrequency,
00450     double widthFrequency,
00451     double stopBandDb) {
00452     if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
00453     BandStopBase::setup (reqOrder,
00454         centerFrequency,
00455         widthFrequency,
00456         stopBandDb);
00457 }
00458 };
00459
00465 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00466 struct LowShelf : PoleFilter <LowShelfBase, StateType, FilterOrder>
00467 {
00475     void setup (double sampleRate,
00476         double cutoffFrequency,
00477         double gainDb,
00478         double stopBandDb) {
00479         LowShelfBase::setup (FilterOrder,
00480             cutoffFrequency / sampleRate,
00481             gainDb,
00482             stopBandDb);
00483     }
00484
00493     void setup (int reqOrder,
00494         double sampleRate,
00495         double cutoffFrequency,
00496         double gainDb,
00497         double stopBandDb) {
00498         if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
00499         LowShelfBase::setup (reqOrder,
00500             cutoffFrequency / sampleRate,
00501             gainDb,
00502             stopBandDb);
00503     }
00504
00505
00506
00507
00508
00515 void setupN(double cutoffFrequency,
00516     double gainDb,
00517     double stopBandDb) {
00518     LowShelfBase::setup (FilterOrder,
00519         cutoffFrequency,
00520         gainDb,
00521         stopBandDb);
00522 }
00523
00531 void setupN(int reqOrder,
00532     double cutoffFrequency,
00533     double gainDb,
00534     double stopBandDb) {
00535     if (reqOrder > FilterOrder) throw_invalid_argument(orderTooHigh);
00536     LowShelfBase::setup (reqOrder,
00537         cutoffFrequency,
00538         gainDb,
00539         stopBandDb);
00540 }
00541
00542 };
00543
00549 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00550 struct HighShelf : PoleFilter <HighShelfBase, StateType, FilterOrder>
00551 {

```

```

00559     void setup (double sampleRate,
00560                 double cutoffFrequency,
00561                 double gainDb,
00562                 double stopBandDb) {
00563         HighShelfBase::setup (FilterOrder,
00564                               cutoffFrequency / sampleRate,
00565                               gainDb,
00566                               stopBandDb);
00567     }
00568
00577     void setup (int reqOrder,
00578                 double sampleRate,
00579                 double cutoffFrequency,
00580                 double gainDb,
00581                 double stopBandDb) {
00582         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00583         HighShelfBase::setup (reqOrder,
00584                               cutoffFrequency / sampleRate,
00585                               gainDb,
00586                               stopBandDb);
00587     }
00588
00589
00590
00591
00592
00593
00600     void setupN(double cutoffFrequency,
00601                 double gainDb,
00602                 double stopBandDb) {
00603         HighShelfBase::setup (FilterOrder,
00604                               cutoffFrequency,
00605                               gainDb,
00606                               stopBandDb);
00607     }
00608
00616     void setupN(int reqOrder,
00617                 double cutoffFrequency,
00618                 double gainDb,
00619                 double stopBandDb) {
00620         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00621         HighShelfBase::setup (reqOrder,
00622                               cutoffFrequency,
00623                               gainDb,
00624                               stopBandDb);
00625     }
00626
00627 };
00628
00634 template <int FilterOrder = DEFAULT_FILTER_ORDER, class StateType = DEFAULT_STATE>
00635 struct BandShelf : PoleFilter <BandShelfBase, StateType, FilterOrder, FilterOrder*2>
00636 {
00645     void setup (double sampleRate,
00646                 double centerFrequency,
00647                 double widthFrequency,
00648                 double gainDb,
00649                 double stopBandDb) {
00650         BandShelfBase::setup (FilterOrder,
00651                               centerFrequency / sampleRate,
00652                               widthFrequency / sampleRate,
00653                               gainDb,
00654                               stopBandDb);
00655     }
00656
00657
00667     void setup (int reqOrder,
00668                 double sampleRate,
00669                 double centerFrequency,
00670                 double widthFrequency,
00671                 double gainDb,
00672                 double stopBandDb) {
00673         if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00674         BandShelfBase::setup (reqOrder,
00675                               centerFrequency / sampleRate,
00676                               widthFrequency / sampleRate,
00677                               gainDb,
00678                               stopBandDb);
00679     }
00680
00681
00682
00683
00684
00685
00686
00694     void setupN(double centerFrequency,
00695                 double widthFrequency,

```

```

00696         double gainDb,
00697         double stopBandDb) {
00698     BandShelfBase::setup (FilterOrder,
00699                          centerFrequency,
00700                          widthFrequency,
00701                          gainDb,
00702                          stopBandDb);
00703 }
00704
00705
00714 void setupN(int reqOrder,
00715            double centerFrequency,
00716            double widthFrequency,
00717            double gainDb,
00718            double stopBandDb) {
00719     if (reqOrder > FilterOrder) throw_invalid_argument (orderTooHigh);
00720     BandShelfBase::setup (reqOrder,
00721                          centerFrequency,
00722                          widthFrequency,
00723                          gainDb,
00724                          stopBandDb);
00725 }
00726
00727
00728 };
00729
00730 }
00731
00732 }
00733
00734 #endif

```

8.7 Common.h

```

00001
00036 #ifndef IIR1_COMMON_H
00037 #define IIR1_COMMON_H
00038
00039 //
00040 // This must be the first file included in every DspFilters header and source
00041 //
00042
00043 #ifdef _MSC_VER
00044 # pragma warning (disable: 4100)
00045 # ifndef _CRT_SECURE_NO_WARNINGS
00046 #   define _CRT_SECURE_NO_WARNINGS
00047 # endif
00048 #endif
00049
00050 // This exports the classes/structures to the windows DLL
00051 #if defined(_WIN32) && defined(iir_EXPORTS)
00052 #define IIR_EXPORT __declspec( dllexport )
00053 #else
00054 #define IIR_EXPORT
00055 #endif
00056
00057 #include <stdlib.h>
00058
00059 #include <cassert>
00060 #include <cfloat>
00061 #include <cmath>
00062 #include <complex>
00063 #include <cstring>
00064 #include <string>
00065 #include <limits>
00066 #include <vector>
00067 #include <stdexcept> // for invalid_argument
00068
00069 static const char orderTooHigh[] = "Requested order is too high. Provide a higher order for the
template.";
00070
00071 #define DEFAULT_FILTER_ORDER 4
00072
00073 inline void throw_invalid_argument(const char* msg) {
00074
00075
00080 #ifndef IIR1_NO_EXCEPTIONS
00081     throw std::invalid_argument (msg);
00082 #else
00083     (void) msg; // Discard parameter
00084     abort();
00085 #endif
00086
00087 }
00088
00089 #endif

```

8.8 Custom.h

```

00001
00036 #ifndef IIR1_CUSTOM_H
00037 #define IIR1_CUSTOM_H
00038
00039 #include "Common.h"
00040 #include "Biquad.h"
00041 #include "Cascade.h"
00042 #include "PoleFilter.h"
00043 #include "State.h"
00044
00045
00046 namespace Iir {
00047
00053 namespace Custom {
00054
00061 struct OnePole : public Biquad
00062 {
00063     void setup (double scale,
00064                double pole,
00065                double zero);
00066 };
00067
00075 struct TwoPole : public Biquad
00076 {
00077     void setup (double scale,
00078                double poleRho,
00079                double poleTheta,
00080                double zeroRho,
00081                double zeroTheta);
00082 };
00083
00089 template <int NSOS, class StateType = DEFAULT_STATE>
00090 struct SOSCascade : CascadeStages<NSOS, StateType>
00091 {
00096     SOSCascade() = default;
00107     SOSCascade(const double (&sosCoefficients)[NSOS][6]) {
00108         CascadeStages<NSOS, StateType>::setup(sosCoefficients);
00109     }
00120     void setup (const double (&sosCoefficients)[NSOS][6]) {
00121         CascadeStages<NSOS, StateType>::setup(sosCoefficients);
00122     }
00123 };
00124
00125 }
00126
00127 }
00128
00129 #endif

```

8.9 Layout.h

```

00001
00036 #ifndef IIR1_LAYOUT_H
00037 #define IIR1_LAYOUT_H
00038
00039 #include "Common.h"
00040 #include "MathSupplement.h"
00041
00047 namespace Iir {
00048
00049     static const char errPoleisNaN[] = "Pole to add is NaN.";
00050     static const char errZeroisNaN[] = "Zero to add is NaN.";
00051
00052     static const char errCantAdd2ndOrder[] = "Can't add 2nd order after a 1st order filter.";
00053
00054     static const char errPolesNotComplexConj[] = "Poles not complex conjugate.";
00055     static const char errZerosNotComplexConj[] = "Zeros not complex conjugate.";
00056
00057     static const char pairIndexOutOfBounds[] = "Pair index out of bounds.";
00058
00062     class IIR_EXPORT LayoutBase
00063     {
00064     public:
00065         LayoutBase ()
00066             : m_numPoles (0)
00067             , m_maxPoles (0)
00068             , m_pair (nullptr)
00069         {
00070         }
00071
00072         LayoutBase (int maxPoles, PoleZeroPair* pairs)
00073             : m_numPoles (0)
00074             , m_maxPoles (maxPoles)
00075             , m_pair (pairs)

```

```

00076     {
00077     }
00078
00079     void setStorage (const LayoutBase& other)
00080     {
00081         m_numPoles = 0;
00082         m_maxPoles = other.m_maxPoles;
00083         m_pair = other.m_pair;
00084     }
00085
00086     void reset ()
00087     {
00088         m_numPoles = 0;
00089     }
00090
00091     int getNumPoles () const
00092     {
00093         return m_numPoles;
00094     }
00095
00096     int getMaxPoles () const
00097     {
00098         return m_maxPoles;
00099     }
00100
00101     void add (const complex_t& pole, const complex_t& zero)
00102     {
00103         if (m_numPoles&1)
00104             throw_invalid_argument (errCantAdd2ndOrder);
00105         if (Iir::is_nan(pole))
00106             throw_invalid_argument (errPoleisNaN);
00107         if (Iir::is_nan(zero))
00108             throw_invalid_argument (errZeroisNaN);
00109         m_pair[m_numPoles/2] = PoleZeroPair (pole, zero);
00110         ++m_numPoles;
00111     }
00112
00113     void addPoleZeroConjugatePairs (const complex_t& pole,
00114                                     const complex_t& zero)
00115     {
00116         if (m_numPoles&1)
00117             throw_invalid_argument (errCantAdd2ndOrder);
00118         if (Iir::is_nan(pole))
00119             throw_invalid_argument (errPoleisNaN);
00120         if (Iir::is_nan(zero))
00121             throw_invalid_argument (errZeroisNaN);
00122         m_pair[m_numPoles/2] = PoleZeroPair (
00123             pole, zero, std::conj (pole), std::conj (zero));
00124         m_numPoles += 2;
00125     }
00126
00127     void add (const ComplexPair& poles, const ComplexPair& zeros)
00128     {
00129         if (m_numPoles&1)
00130             throw_invalid_argument (errCantAdd2ndOrder);
00131         if (!poles.isMatchedPair ())
00132             throw_invalid_argument (errPolesNotComplexConj);
00133         if (!zeros.isMatchedPair ())
00134             throw_invalid_argument (errZerosNotComplexConj);
00135         m_pair[m_numPoles/2] = PoleZeroPair (poles.first, zeros.first,
00136                                             poles.second, zeros.second);
00137         m_numPoles += 2;
00138     }
00139
00140     const PoleZeroPair& getPair (int pairIndex) const
00141     {
00142         if ((pairIndex < 0) || (pairIndex >= (m_numPoles+1)/2))
00143             throw_invalid_argument (pairIndexOutOfBounds);
00144         return m_pair[pairIndex];
00145     }
00146
00147     const PoleZeroPair& operator[] (int pairIndex) const
00148     {
00149         return getPair (pairIndex);
00150     }
00151
00152     double getNormalW () const
00153     {
00154         return m_normalW;
00155     }
00156
00157     double getNormalGain () const
00158     {
00159         return m_normalGain;
00160     }
00161
00162     void setNormal (double w, double g)

```

```

00163         {
00164             m_normalW = w;
00165             m_normalGain = g;
00166         }
00167
00168     private:
00169         int m_numPoles = 0;
00170         int m_maxPoles = 0;
00171         PoleZeroPair* m_pair = nullptr;
00172         double m_normalW = 0;
00173         double m_normalGain = 1;
00174     };
00175
00176 //-----
00177
00181     template <int MaxPoles>
00182     class Layout
00183     {
00184     public:
00185         operator LayoutBase ()
00186         {
00187             return LayoutBase (MaxPoles, m_pairs);
00188         }
00189
00190     private:
00191         PoleZeroPair m_pairs[(MaxPoles+1)/2] = {};
00192     };
00193 }
00194 }
00195
00196 #endif

```

8.10 MathSupplement.h

```

00001
00036 #ifndef IIR1_MATHSUPPLEMENT_H
00037 #define IIR1_MATHSUPPLEMENT_H
00038
00039 #include "Common.h"
00040
00041 #include <complex>
00042
00043 #ifdef _MSC_VER
00044     // Under Unix these have already default instantiations but not under Vis Studio
00045     template class IIR_EXPORT std::complex<double>;
00046     template class IIR_EXPORT std::complex<float>;
00047 #endif
00048
00049 namespace Iir {
00050
00051     const double doublePi = 3.1415926535897932384626433832795028841971;
00052     const double doublePi_2 = 1.5707963267948966192313216916397514420986;
00053     const double doubleLn2 = 0.69314718055994530941723212145818;
00054     const double doubleLn10 = 2.3025850929940456840179914546844;
00055
00056     typedef std::complex<double> complex_t;
00057     typedef std::pair<complex_t, complex_t> complex_pair_t;
00058
00059     inline const complex_t infinity()
00060     {
00061         return complex_t (std::numeric_limits<double>::infinity());
00062     }
00063
00064     template <typename Ty, typename To>
00065     inline std::complex<Ty> addmul (const std::complex<Ty>& c,
00066                                     Ty v,
00067                                     const std::complex<To>& c1)
00068     {
00069         return std::complex <Ty> (
00070             c.real() + v * c1.real(), c.imag() + v * c1.imag());
00071     }
00072
00073     template <typename Ty>
00074     inline Ty asinh (Ty x)
00075     {
00076         return log (x + std::sqrt (x * x + 1 ));
00077     }
00078
00079     template <typename Ty>
00080     inline bool is_nan (Ty v)
00081     {
00082         return !(v == v);
00083     }
00084
00085     template <>
00086     inline bool is_nan<complex_t> (complex_t v)

```

```

00087 {
00088     return Iir::is_nan (v.real()) || Iir::is_nan (v.imag());
00089 }
00090
00091 }
00092
00093 #endif

```

8.11 PoleFilter.h

```

00001
00036 #ifndef IIR1_POLEFILTER_H
00037 #define IIR1_POLEFILTER_H
00038
00039 #include "Common.h"
00040 #include "MathSupplement.h"
00041 #include "Cascade.h"
00042 #include "State.h"
00043
00044 // Purely for debugging...
00045 #include <iostream>
00046
00047 namespace Iir {
00048
00049 /**
00050  * Base for filters designed via algorithmic placement of poles and zeros.
00051  *
00052  * Typically, the filter is first designed as a half-band low pass or
00053  * low shelf analog filter (s-plane). Then, using a transformation such
00054  * as the ones from Constantinides, the poles and zeros of the analog filter
00055  * are calculated in the z-plane.
00056  *
00057  */
00058
00062     class IIR_EXPORT PoleFilterBase2 : public Cascade
00063     {
00064     public:
00065         // This gets the poles/zeros directly from the digital
00066         // prototype. It is used to double check the correctness
00067         // of the recovery of pole/zeros from biquad coefficients.
00068         //
00069         // It can also be used to accelerate the interpolation
00070         // of pole/zeros for parameter modulation, since a pole
00071         // filter already has them calculated
00072
00073         PoleFilterBase2() = default;
00074
00075         std::vector<PoleZeroPair> getPoleZeros () const
00076         {
00077             std::vector<PoleZeroPair> vpz;
00078             const int pairs = (m_digitalProto.getNumPoles () + 1) / 2;
00079             for (int i = 0; i < pairs; ++i)
00080                 vpz.push_back (m_digitalProto[i]);
00081             return vpz;
00082         }
00083
00084     protected:
00085         LayoutBase m_digitalProto = {};
00086     };
00087
00088
00093     template <class AnalogPrototype>
00094     class PoleFilterBase : public PoleFilterBase2
00095     {
00096     protected:
00097         void setPrototypeStorage (const LayoutBase& analogStorage,
00098                                 const LayoutBase& digitalStorage)
00099         {
00100             m_analogProto.setStorage (analogStorage);
00101             m_digitalProto = digitalStorage;
00102         }
00103
00104     protected:
00105         AnalogPrototype m_analogProto = {};
00106     };
00107
00108 //-----
00109
00113     template <class BaseClass,
00114              class StateType,
00115              int MaxAnalogPoles,
00116              int MaxDigitalPoles = MaxAnalogPoles>
00117     struct PoleFilter : BaseClass
00118     {
00119     public:
00120         , CascadeStages <(MaxDigitalPoles + 1) / 2 , StateType>

```

```

00121         PoleFilter ()
00122         {
00123             // This glues together the factored base classes
00124             // with the templated storage classes.
00125             BaseClass::setCascadeStorage (this->getCascadeStorage());
00126             BaseClass::setPrototypeStorage (m_analogStorage, m_digitalStorage);
00127             CascadeStages<(MaxDigitalPoles + 1) / 2 , StateType>::reset();
00128         }
00129
00130         PoleFilter(const PoleFilter&) = default;
00131
00132         PoleFilter& operator=(const PoleFilter&)
00133         {
00134             // Reset the filter state when copied for now
00135             CascadeStages<(MaxDigitalPoles + 1) / 2 , StateType>::reset();
00136             return *this;
00137         }
00138
00139     private:
00140         Layout <MaxAnalogPoles> m_analogStorage = {};
00141         Layout <MaxDigitalPoles> m_digitalStorage = {};
00142     };
00143
00144 //-----
00145
00160     class IIR_EXPORT LowPassTransform
00161     {
00162     public:
00163         LowPassTransform (double fc,
00164                         LayoutBase& digital,
00165                         LayoutBase const& analog);
00166
00167     private:
00168         complex_t transform (complex_t c);
00169
00170         double f = 0.0;
00171     };
00172
00173 //-----
00174
00178     class IIR_EXPORT HighPassTransform
00179     {
00180     public:
00181         HighPassTransform (double fc,
00182                         LayoutBase& digital,
00183                         LayoutBase const& analog);
00184
00185     private:
00186         complex_t transform (complex_t c);
00187
00188         double f = 0.0;
00189     };
00190
00191 //-----
00192
00196     class IIR_EXPORT BandPassTransform
00197     {
00198
00199     public:
00200         BandPassTransform (double fc,
00201                         double fw,
00202                         LayoutBase& digital,
00203                         LayoutBase const& analog);
00204
00205     private:
00206         ComplexPair transform (complex_t c);
00207
00208         double wc = 0.0;
00209         double wc2 = 0.0;
00210         double a = 0.0;
00211         double b = 0.0;
00212         double a2 = 0.0;
00213         double b2 = 0.0;
00214         double ab = 0.0;
00215         double ab_2 = 0.0;
00216     };
00217
00218 //-----
00219
00223     class IIR_EXPORT BandStopTransform
00224     {
00225     public:
00226         BandStopTransform (double fc,
00227                         double fw,
00228                         LayoutBase& digital,
00229                         LayoutBase const& analog);
00230

```

```

00231     private:
00232     ComplexPair transform (complex_t c);
00233
00234     double wc = 0.0;
00235     double wc2 = 0.0;
00236     double a = 0.0;
00237     double b = 0.0;
00238     double a2 = 0.0;
00239     double b2 = 0.0;
00240 };
00241
00242 }
00243
00244 #endif

```

8.12 RBJ.h

```

00001
00036 #ifndef IIR1_RBJ_H
00037 #define IIR1_RBJ_H
00038
00039 #include "Common.h"
00040 #include "Biquad.h"
00041 #include "State.h"
00042
00043 namespace Iir {
00044
00059 #define ONESQRT2 (1/sqrt(2))
00060
00061 namespace RBJ {
00062
00066     struct IIR_EXPORT RBJbase : Biquad
00067     {
00068     public:
00070         template <typename Sample>
00071         inline Sample filter(Sample s) {
00072             return static_cast<Sample>(state.filter((double)s, *this));
00073         }
00075         void reset() {
00076             state.reset();
00077         }
00079         const DirectFormI& getState() {
00080             return state;
00081         }
00082     private:
00083         DirectFormI state = {};
00084     };
00085
00089     struct IIR_EXPORT LowPass : RBJbase
00090     {
00096         void setupN(double cutoffFrequency,
00097                     double q = ONESQRT2);
00105         void setup(double sampleRate,
00106                     double cutoffFrequency,
00107                     double q = ONESQRT2) {
00108             setupN(cutoffFrequency / sampleRate, q);
00109         }
00110     };
00111
00115     struct IIR_EXPORT HighPass : RBJbase
00116     {
00122         void setupN(double cutoffFrequency,
00123                     double q = ONESQRT2);
00130         void setup(double sampleRate,
00131                     double cutoffFrequency,
00132                     double q = ONESQRT2) {
00133             setupN(cutoffFrequency / sampleRate, q);
00134         }
00135     };
00136
00140     struct IIR_EXPORT BandPass1 : RBJbase
00141     {
00147         void setupN(double centerFrequency,
00148                     double bandwidth);
00155         void setup(double sampleRate,
00156                     double centerFrequency,
00157                     double bandwidth) {
00158             setupN(centerFrequency / sampleRate, bandwidth);
00159         }
00160     };
00161
00165     struct IIR_EXPORT BandPass2 : RBJbase
00166     {
00172         void setupN(double centerFrequency,
00173                     double bandwidth);

```

```
00180         void setup (double sampleRate,
00181                     double centerFrequency,
00182                     double bandWidth) {
00183             setupN(centerFrequency / sampleRate, bandWidth);
00184         }
00185     };
00186
00191     struct IIR_EXPORT BandStop : RBJbase
00192     {
00193         void setupN(double centerFrequency,
00194                     double bandWidth);
00195         void setup (double sampleRate,
00206                     double centerFrequency,
00207                     double bandWidth) {
00208             setupN(centerFrequency / sampleRate, bandWidth);
00209         }
00210     };
00211
00212
00224     struct IIR_EXPORT IIRNotch : RBJbase
00225     {
00231         void setupN(double centerFrequency,
00232                     double q_factor = 10);
00233         void setup (double sampleRate,
00240                     double centerFrequency,
00241                     double q_factor = 10) {
00242             setupN(centerFrequency / sampleRate, q_factor);
00243         }
00244     };
00245
00249     struct IIR_EXPORT LowShelf : RBJbase
00250     {
00257         void setupN(double cutoffFrequency,
00258                     double gainDb,
00259                     double shelfSlope = 1);
00267         void setup (double sampleRate,
00268                     double cutoffFrequency,
00269                     double gainDb,
00270                     double shelfSlope = 1) {
00271             setupN( cutoffFrequency / sampleRate, gainDb, shelfSlope);
00272         }
00273     };
00274
00278     struct IIR_EXPORT HighShelf : RBJbase
00279     {
00286         void setupN(double cutoffFrequency,
00287                     double gainDb,
00288                     double shelfSlope = 1);
00296         void setup (double sampleRate,
00297                     double cutoffFrequency,
00298                     double gainDb,
00299                     double shelfSlope = 1) {
00300             setupN( cutoffFrequency / sampleRate, gainDb, shelfSlope);
00301         }
00302     };
00303
00307     struct IIR_EXPORT BandShelf : RBJbase
00308     {
00315         void setupN(double centerFrequency,
00316                     double gainDb,
00317                     double bandWidth);
00325         void setup (double sampleRate,
00326                     double centerFrequency,
00327                     double gainDb,
00328                     double bandWidth) {
00329             setupN(centerFrequency / sampleRate, gainDb, bandWidth);
00330         }
00331     };
00332
00336     struct IIR_EXPORT AllPass : RBJbase
00337     {
00343         void setupN(double phaseFrequency,
00344                     double q = ONESQRT2);
00345
00352         void setup (double sampleRate,
00353                     double phaseFrequency,
00354                     double q = ONESQRT2) {
00355             setupN( phaseFrequency / sampleRate, q);
00356         }
00357     };
00358
00359 }
00360
00361 }
00362
00363
00364 #endif
```

8.13 State.h

```

00001
00036 #ifndef IIR1_STATE_H
00037 #define IIR1_STATE_H
00038
00039 #include "Common.h"
00040 #include "Biquad.h"
00041
00042
00043 #define DEFAULT_STATE DirectFormII
00044
00045 namespace Iir {
00046
00055     class IIR_EXPORT DirectFormI
00056     {
00057     public:
00058         DirectFormI () = default;
00059
00060         void reset ()
00061         {
00062             m_x1 = 0;
00063             m_x2 = 0;
00064             m_y1 = 0;
00065             m_y2 = 0;
00066         }
00067
00068         inline double filter(const double in,
00069                             const Biquad& s)
00070         {
00071             const double out = s.m_b0*in + s.m_b1*m_x1 + s.m_b2*m_x2
00072                             - s.m_a1*m_y1 - s.m_a2*m_y2;
00073             m_x2 = m_x1;
00074             m_y2 = m_y1;
00075             m_x1 = in;
00076             m_y1 = out;
00077
00078             return out;
00079         }
00080
00081     protected:
00082         double m_x2 = 0.0; // x[n-2]
00083         double m_y2 = 0.0; // y[n-2]
00084         double m_x1 = 0.0; // x[n-1]
00085         double m_y1 = 0.0; // y[n-1]
00086     };
00087
00088 //-----
00089
00099     class IIR_EXPORT DirectFormII
00100     {
00101     public:
00102         DirectFormII () = default;
00103
00104         void reset ()
00105         {
00106             m_v1 = 0.0;
00107             m_v2 = 0.0;
00108         }
00109
00110         inline double filter(const double in,
00111                             const Biquad& s)
00112         {
00113             const double w = in - s.m_a1*m_v1 - s.m_a2*m_v2;
00114             const double out = s.m_b0*w + s.m_b1*m_v1 + s.m_b2*m_v2;
00115
00116             m_v2 = m_v1;
00117             m_v1 = w;
00118
00119             return out;
00120         }
00121
00122     private:
00123         double m_v1 = 0.0; // v[-1]
00124         double m_v2 = 0.0; // v[-2]
00125     };
00126
00127 //-----
00128
00129
00130     class IIR_EXPORT TransposedDirectFormII
00131     {
00132     public:
00133         TransposedDirectFormII () = default;
00134
00135         void reset ()
00136         {

```

```

00137         m_s1 = 0.0;
00138         m_s1_1 = 0.0;
00139         m_s2 = 0.0;
00140         m_s2_1 = 0.0;
00141     }
00142
00143     inline double filter(const double in,
00144                         const Biquad& s)
00145     {
00146         const double out = m_s1_1 + s.m_b0*in;
00147         m_s1 = m_s2_1 + s.m_b1*in - s.m_a1*out;
00148         m_s2 = s.m_b2*in - s.m_a2*out;
00149         m_s1_1 = m_s1;
00150         m_s2_1 = m_s2;
00151
00152         return out;
00153     }
00154
00155 private:
00156     double m_s1 = 0.0;
00157     double m_s1_1 = 0.0;
00158     double m_s2 = 0.0;
00159     double m_s2_1 = 0.0;
00160 };
00161
00162 }
00163
00164 #endif

```

8.14 Types.h

```

00001
00036 #ifndef IIR1_TYPES_H
00037 #define IIR1_TYPES_H
00038
00039 #include "Common.h"
00040 #include "MathSupplement.h"
00041
00042 namespace Iir {
00043
00047     struct IIR_EXPORT ComplexPair : complex_pair_t
00048     {
00049         ComplexPair() = default;
00050
00051         explicit ComplexPair (const complex_t& c1)
00052             : complex_pair_t (c1, 0.)
00053         {
00054             if (!isReal()) throw_invalid_argument("A single complex number needs to be
00055 real.");
00056         }
00057
00058         ComplexPair (const complex_t& c1,
00059                     const complex_t& c2)
00060             : complex_pair_t (c1, c2)
00061         {
00062         }
00063
00064         bool isReal () const
00065         {
00066             return first.imag() == 0 && second.imag() == 0;
00067         }
00072
00073         bool isMatchedPair () const
00074         {
00075             if (first.imag() != 0)
00076                 return second == std::conj (first);
00077             else
00078                 return second.imag () == 0 &&
00079                        second.real () != 0 &&
00080                        first.real () != 0;
00081         }
00082
00083         bool is_nan () const
00084         {
00085             return Iir::is_nan (first) || Iir::is_nan (second);
00086         }
00087     };
00088
00092     struct IIR_EXPORT PoleZeroPair
00093     {
00094         ComplexPair poles = ComplexPair();
00095         ComplexPair zeros = ComplexPair();
00096
00097         PoleZeroPair () = default;
00098     };

```

```
00099         // single pole/zero
00100     PoleZeroPair (const complex_t& p, const complex_t& z)
00101         : poles (p), zeros (z)
00102     {
00103     }
00104
00105     // pole/zero pair
00106     PoleZeroPair (const complex_t& p1, const complex_t& z1,
00107                  const complex_t& p2, const complex_t& z2)
00108         : poles (p1, p2)
00109         , zeros (z1, z2)
00110     {
00111     }
00112
00113     bool isSinglePole () const
00114     {
00115         return poles.second == 0. && zeros.second == 0.;
00116     }
00117
00118     bool is_nan () const
00119     {
00120         return poles.is_nan() || zeros.is_nan();
00121     }
00122 };
00123
00124
00125 }
00126
00127 #endif
```


Index

- applyScale
 - lir::Biquad, [46](#)
- DSP IIR Realtime C++ filter library, [1](#)
- filter
 - lir::Biquad, [47](#)
 - lir::CascadeStages< MaxStages, StateType >, [51](#)
- getA0
 - lir::Biquad, [47](#)
- getA1
 - lir::Biquad, [47](#)
- getA2
 - lir::Biquad, [47](#)
- getB0
 - lir::Biquad, [47](#)
- getB1
 - lir::Biquad, [47](#)
- getB2
 - lir::Biquad, [47](#)
- getCascadeStorage
 - lir::CascadeStages< MaxStages, StateType >, [51](#)
- getNumStages
 - lir::Cascade, [50](#)
- getPoleZeros
 - lir::Biquad, [47](#)
 - lir::Cascade, [50](#)
- lir, [8](#)
- iir/Biquad.h, [100](#)
- iir/Butterworth.h, [101](#)
- iir/Cascade.h, [105](#)
- iir/ChebyshevI.h, [106](#)
- iir/ChebyshevII.h, [112](#)
- iir/Common.h, [117](#)
- iir/Custom.h, [118](#)
- iir/Layout.h, [118](#)
- iir/MathSupplement.h, [120](#)
- iir/PoleFilter.h, [121](#)
- iir/RBJ.h, [123](#)
- iir/State.h, [125](#)
- iir/Types.h, [126](#)
- lir::BandPassTransform, [26](#)
- lir::BandStopTransform, [45](#)
- lir::Biquad, [45](#)
 - applyScale, [46](#)
 - filter, [47](#)
 - getA0, [47](#)
 - getA1, [47](#)
 - getA2, [47](#)
 - getB0, [47](#)
 - getB1, [47](#)
 - getB2, [47](#)
 - getPoleZeros, [47](#)
 - response, [47](#)
 - setCoefficients, [48](#)
 - setIdentity, [48](#)
 - setOnePole, [48](#)
 - setPoleZeroPair, [48](#)
 - setTwoPole, [48](#)
- lir::BiquadPoleState, [49](#)
- lir::Butterworth, [11](#)
 - lir::Butterworth::AnalogLowPass, [14](#)
 - lir::Butterworth::AnalogLowShelf, [15](#)
 - lir::Butterworth::BandPass< FilterOrder, StateType >, [16](#)
 - setup, [16](#), [17](#)
 - setupN, [17](#)
 - lir::Butterworth::BandPassBase, [25](#)
 - lir::Butterworth::BandShelf< FilterOrder, StateType >, [26](#)
 - setup, [27](#)
 - setupN, [28](#)
 - lir::Butterworth::BandShelfBase, [35](#)
 - lir::Butterworth::BandStop< FilterOrder, StateType >, [36](#)
 - setup, [37](#)
 - setupN, [37](#), [38](#)
 - lir::Butterworth::BandStopBase, [44](#)
 - lir::Butterworth::HighPass< FilterOrder, StateType >, [53](#)
 - setup, [54](#)
 - setupN, [54](#)
 - lir::Butterworth::HighPassBase, [60](#)
 - lir::Butterworth::HighShelf< FilterOrder, StateType >, [62](#)
 - setup, [62](#), [63](#)
 - setupN, [63](#)
 - lir::Butterworth::HighShelfBase, [69](#)
 - lir::Butterworth::LowPass< FilterOrder, StateType >, [73](#)
 - setup, [73](#), [74](#)
 - setupN, [74](#)
 - lir::Butterworth::LowPassBase, [80](#)
 - lir::Butterworth::LowShelf< FilterOrder, StateType >, [82](#)
 - setup, [82](#), [83](#)
 - setupN, [83](#)
 - lir::Butterworth::LowShelfBase, [89](#)
- lir::Cascade, [49](#)
 - getNumStages, [50](#)
 - getPoleZeros, [50](#)
 - operator[], [50](#)
 - response, [50](#)
- lir::Cascade::Storage, [98](#)
- lir::CascadeStages< MaxStages, StateType >, [50](#)
 - filter, [51](#)
 - getCascadeStorage, [51](#)
 - reset, [51](#)
 - setup, [52](#)
- lir::ChebyshevI, [11](#)
- lir::ChebyshevI::AnalogLowPass, [14](#)

- lir::ChebyshevI::AnalogLowShelf, [15](#)
- lir::ChebyshevI::BandPass< FilterOrder, StateType >, [18](#)
 - setup, [18](#), [19](#)
 - setupN, [19](#)
- lir::ChebyshevI::BandPassBase, [25](#)
- lir::ChebyshevI::BandShelf< FilterOrder, StateType >, [28](#)
 - setup, [29](#), [30](#)
 - setupN, [30](#)
- lir::ChebyshevI::BandShelfBase, [35](#)
- lir::ChebyshevI::BandStop< FilterOrder, StateType >, [38](#)
 - setup, [39](#)
 - setupN, [39](#), [40](#)
- lir::ChebyshevI::BandStopBase, [44](#)
- lir::ChebyshevI::HighPass< FilterOrder, StateType >, [55](#)
 - setup, [55](#), [56](#)
 - setupN, [56](#)
- lir::ChebyshevI::HighPassBase, [60](#)
- lir::ChebyshevI::HighShelf< FilterOrder, StateType >, [64](#)
 - setup, [64](#), [65](#)
 - setupN, [65](#)
- lir::ChebyshevI::HighShelfBase, [70](#)
- lir::ChebyshevI::LowPass< FilterOrder, StateType >, [74](#)
 - setup, [75](#)
 - setupN, [76](#)
- lir::ChebyshevI::LowPassBase, [80](#)
- lir::ChebyshevI::LowShelf< FilterOrder, StateType >, [84](#)
 - setup, [84](#), [85](#)
 - setupN, [85](#)
- lir::ChebyshevI::LowShelfBase, [90](#)
- lir::ChebyshevII, [12](#)
- lir::ChebyshevII::AnalogLowPass, [14](#)
- lir::ChebyshevII::AnalogLowShelf, [15](#)
- lir::ChebyshevII::BandPass< FilterOrder, StateType >, [20](#)
 - setup, [20](#), [21](#)
 - setupN, [21](#)
- lir::ChebyshevII::BandPassBase, [26](#)
- lir::ChebyshevII::BandShelf< FilterOrder, StateType >, [31](#)
 - setup, [31](#), [32](#)
 - setupN, [32](#)
- lir::ChebyshevII::BandShelfBase, [36](#)
- lir::ChebyshevII::BandStop< FilterOrder, StateType >, [40](#)
 - setup, [41](#)
 - setupN, [41](#), [42](#)
- lir::ChebyshevII::BandStopBase, [45](#)
- lir::ChebyshevII::HighPass< FilterOrder, StateType >, [56](#)
 - setup, [57](#)
 - setupN, [58](#)
- lir::ChebyshevII::HighPassBase, [61](#)
- lir::ChebyshevII::HighShelf< FilterOrder, StateType >, [66](#)
 - setup, [66](#), [67](#)
 - setupN, [67](#)
- lir::ChebyshevII::HighShelfBase, [70](#)
- lir::ChebyshevII::LowPass< FilterOrder, StateType >, [76](#)
 - setup, [77](#)
 - setupN, [78](#)
- lir::ChebyshevII::LowPassBase, [81](#)
- lir::ChebyshevII::LowShelf< FilterOrder, StateType >, [86](#)
 - setup, [86](#), [87](#)
 - setupN, [87](#)
- lir::ChebyshevII::LowShelfBase, [90](#)
- lir::ComplexPair, [52](#)
 - isMatchedPair, [52](#)
- lir::Custom, [12](#)
- lir::Custom::OnePole, [91](#)
- lir::Custom::SOSCascade< NSOS, StateType >, [97](#)
 - setup, [98](#)
 - SOSCascade, [97](#), [98](#)
- lir::Custom::TwoPole, [98](#)
- lir::DirectFormI, [52](#)
- lir::DirectFormII, [53](#)
- lir::HighPassTransform, [61](#)
- lir::Layout< MaxPoles >, [72](#)
- lir::LayoutBase, [72](#)
- lir::LowPassTransform, [81](#)
- lir::PoleFilter< BaseClass, StateType, MaxAnalogPoles, MaxDigitalPoles >, [93](#)
- lir::PoleFilterBase< AnalogPrototype >, [94](#)
- lir::PoleFilterBase2, [94](#)
- lir::PoleZeroPair, [95](#)
- lir::RBJ::AllPass, [12](#)
 - setup, [13](#)
 - setupN, [13](#)
- lir::RBJ::BandPass1, [22](#)
 - setup, [23](#)
 - setupN, [23](#)
- lir::RBJ::BandPass2, [23](#)
 - setup, [24](#)
 - setupN, [24](#)
- lir::RBJ::BandShelf, [33](#)
 - setup, [34](#)
 - setupN, [34](#)
- lir::RBJ::BandStop, [42](#)
 - setup, [43](#)
 - setupN, [43](#)
- lir::RBJ::HighPass, [58](#)
 - setup, [59](#)
 - setupN, [60](#)
- lir::RBJ::HighShelf, [68](#)
 - setup, [69](#)
 - setupN, [69](#)
- lir::RBJ::IIRNotch, [71](#)
 - setup, [72](#)
 - setupN, [72](#)

- lir::RBJ::LowPass, 78
 - setup, 79
 - setupN, 80
- lir::RBJ::LowShelf, 88
 - setup, 89
 - setupN, 89
- lir::RBJ::RBJbase, 96
- lir::TransposedDirectFormII, 98
- isMatchedPair
 - lir::ComplexPair, 52
- operator[]
 - lir::Cascade, 50
- reset
 - lir::CascadeStages< MaxStages, StateType >, 51
- response
 - lir::Biquad, 47
 - lir::Cascade, 50
- setCoefficients
 - lir::Biquad, 48
- setIdentity
 - lir::Biquad, 48
- setOnePole
 - lir::Biquad, 48
- setPoleZeroPair
 - lir::Biquad, 48
- setTwoPole
 - lir::Biquad, 48
- setup
 - lir::Butterworth::BandPass< FilterOrder, StateType >, 16, 17
 - lir::Butterworth::BandShelf< FilterOrder, StateType >, 27
 - lir::Butterworth::BandStop< FilterOrder, StateType >, 37
 - lir::Butterworth::HighPass< FilterOrder, StateType >, 54
 - lir::Butterworth::HighShelf< FilterOrder, StateType >, 62, 63
 - lir::Butterworth::LowPass< FilterOrder, StateType >, 73, 74
 - lir::Butterworth::LowShelf< FilterOrder, StateType >, 82, 83
 - lir::CascadeStages< MaxStages, StateType >, 52
 - lir::ChebyshevI::BandPass< FilterOrder, StateType >, 18, 19
 - lir::ChebyshevI::BandShelf< FilterOrder, StateType >, 29, 30
 - lir::ChebyshevI::BandStop< FilterOrder, StateType >, 39
 - lir::ChebyshevI::HighPass< FilterOrder, StateType >, 55, 56
 - lir::ChebyshevI::HighShelf< FilterOrder, StateType >, 64, 65
 - lir::ChebyshevI::LowPass< FilterOrder, StateType >, 75
 - lir::ChebyshevI::LowShelf< FilterOrder, StateType >, 84, 85
 - lir::ChebyshevII::BandPass< FilterOrder, StateType >, 20, 21
 - lir::ChebyshevII::BandShelf< FilterOrder, StateType >, 31, 32
 - lir::ChebyshevII::BandStop< FilterOrder, StateType >, 41
 - lir::ChebyshevII::HighPass< FilterOrder, StateType >, 57
 - lir::ChebyshevII::HighShelf< FilterOrder, StateType >, 66, 67
 - lir::ChebyshevII::LowPass< FilterOrder, StateType >, 77
 - lir::ChebyshevII::LowShelf< FilterOrder, StateType >, 86, 87
 - lir::Custom::SOSCascade< NSOS, StateType >, 98
 - lir::RBJ::AllPass, 13
 - lir::RBJ::BandPass1, 23
 - lir::RBJ::BandPass2, 24
 - lir::RBJ::BandShelf, 34
 - lir::RBJ::BandStop, 43
 - lir::RBJ::HighPass, 59
 - lir::RBJ::HighShelf, 69
 - lir::RBJ::IIRNotch, 72
 - lir::RBJ::LowPass, 79
 - lir::RBJ::LowShelf, 89
- setupN
 - lir::Butterworth::BandPass< FilterOrder, StateType >, 17
 - lir::Butterworth::BandShelf< FilterOrder, StateType >, 28
 - lir::Butterworth::BandStop< FilterOrder, StateType >, 37, 38
 - lir::Butterworth::HighPass< FilterOrder, StateType >, 54
 - lir::Butterworth::HighShelf< FilterOrder, StateType >, 63
 - lir::Butterworth::LowPass< FilterOrder, StateType >, 74
 - lir::Butterworth::LowShelf< FilterOrder, StateType >, 83
 - lir::ChebyshevI::BandPass< FilterOrder, StateType >, 19
 - lir::ChebyshevI::BandShelf< FilterOrder, StateType >, 30
 - lir::ChebyshevI::BandStop< FilterOrder, StateType >, 39, 40
 - lir::ChebyshevI::HighPass< FilterOrder, StateType >, 56
 - lir::ChebyshevI::HighShelf< FilterOrder, StateType >, 65
 - lir::ChebyshevI::LowPass< FilterOrder, StateType >, 76
 - lir::ChebyshevI::LowShelf< FilterOrder, StateType >, 85
 - lir::ChebyshevII::BandPass< FilterOrder, StateType >, 21

Type >, [21](#)
lir::ChebyshevII::BandShelf< FilterOrder, State-
 Type >, [32](#)
lir::ChebyshevII::BandStop< FilterOrder, State-
 Type >, [41](#), [42](#)
lir::ChebyshevII::HighPass< FilterOrder, StateType
 >, [58](#)
lir::ChebyshevII::HighShelf< FilterOrder, State-
 Type >, [67](#)
lir::ChebyshevII::LowPass< FilterOrder, StateType
 >, [78](#)
lir::ChebyshevII::LowShelf< FilterOrder, StateType
 >, [87](#)
lir::RBJ::AllPass, [13](#)
lir::RBJ::BandPass1, [23](#)
lir::RBJ::BandPass2, [24](#)
lir::RBJ::BandShelf, [34](#)
lir::RBJ::BandStop, [43](#)
lir::RBJ::HighPass, [60](#)
lir::RBJ::HighShelf, [69](#)
lir::RBJ::IIRNotch, [72](#)
lir::RBJ::LowPass, [80](#)
lir::RBJ::LowShelf, [89](#)
SOSCascade
 lir::Custom::SOSCascade< NSOS, StateType >,
 [97](#), [98](#)