

Audio Equalizer Filtering Library Help

1 Audio Equalization Filtering Library

Files Referenced

Name	Description
audio_equalizer.h	Audio Equalizer (DSP) functions for the PIC32MX and PIC32MZ device families
audio_equalizer_fixedpoint.h	Audio Equalizer (DSP) fixed point typedefs.
GraphicEqualizer6x2_Q15.h	16 Bit Filter definition for 6 Bands, with 2 Filters/Band.
GraphicEqualizer6x2_Q31.h	32 Bit Filter definition for 6 Bands, with 2 Filters/Band.
myFilters4x2_Q15.h	16 Bit Filter definition for 4 Bands, with 2 Filters/Band.
myFilters4x2_Q31.h	32 Bit Filter definition for 4 Bands, with 2 Filters/Band.
myFilters4x3_Q15.h	16 Bit Filter definition for 4 Bands, with 3 Filters/Band.
myFilters4x3_Q31.h	32 Bit Filter definition for 4 Bands, with 3 Filters/Band.
myFilters5x2_Q15.h	16 Bit Filter definition for 5 Bands, with 2 Filters/Band.
myFilters5x2_Q31.h	32 Bit Filter definition for 5 Bands, with 2 Filters/Band.
myFilters6x2_Q15.h	16 Bit Filter definition for 6 Bands, with 2 Filters/Band.
myFilters6x2_Q31.h	32 Bit Filter definition for 6 Bands, with 2 Filters/Band.
myFilters7x2_Q15.h	16 Bit Filter definition for 7 Bands, with 2 Filters/Band.
myFilters7x2_Q31.h	32 Bit Filter definition for 7 Bands, with 2 Filters/Band.
myFilters8x2_Q15.h	16 Bit Filter definition for 8 Bands, with 2 Filters/Band.
myFilters8x2_Q31.h	32 Bit Filter definition for 8 Bands, with 2 Filters/Band.
ParametricFilters1x8_Q15.h	16 Bit Filter definition for an 8 filter chain
ParametricFilters1x8_Q31.h	32 Bit Filter definition for an 8 filter chain
ParametricFilters1x8_Q31_Hacked.h	32 Bit Filter definition for an 8 filter chain, with edits to show 16 bit effects

Description

1.1 Introduction

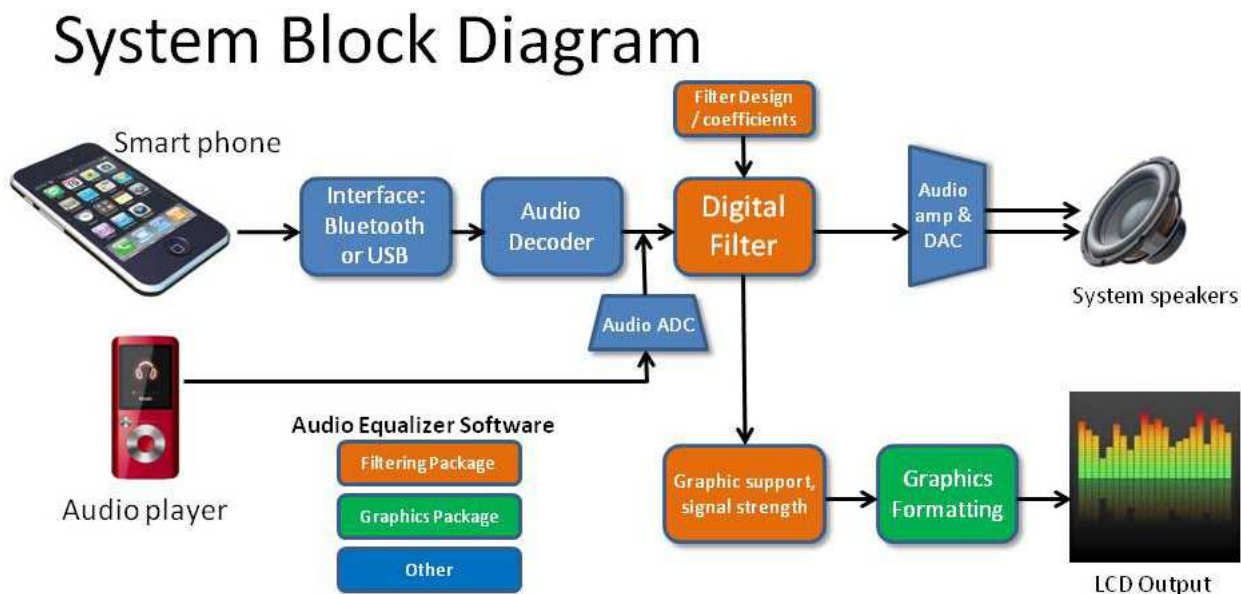
Audio Equalization Filtering Library for Microchip Microcontrollers

This [library](#) provides filtering C and assembly functions for audio equalization using infinite impulse response (IIR) filters. Filter architectures for traditional **graphical equalization** and for **parametric equalization** are supported. Filters can be designed in Matlab (tm), Octave, or any number of dedicated filter design packages. The differences in use and filter structure between graphic equalization filters and parametric equalization filters are discussed in the [Library Overview](#) section below.

The **Graphic Equalizer Display Library** is an adjunct (supporting) [library](#) that is often used with this [library](#). It works on any Microchip device that drives an LCD display, providing Graphic Equalizer displays (see below) for the host application. The "Graphic" part of a graphic equalizer displays signal strength by frequency. Graphic Equalizers are used to adjust the spectral content of music by providing gain or attenuation to parts of the music based on frequency. It works hand-in-hand with the Audio Equalizer Filtering Library, which does the actual signal processing (filtering).

Description

Audio equalization filtering is just one part of an overall system that delivers music to the user:

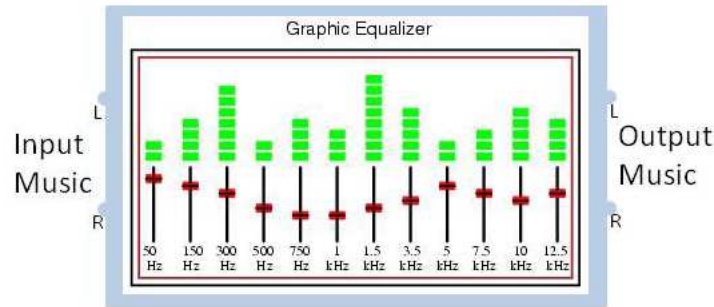


The blocks in **orange** are part of this [library](#). The blocks in **green** are part of the Graphic Equalizer Display [library](#).

As shown above a smart phone can provide music through a Bluetooth or USB interface that is then decoded using the Audio Decoder. Raw left/right samples are then filtered and passed on to an audio amplifier and DAC to convert digital samples into analog sound. The Digital Filter block also measure frequency band signal strength (energy) and passes this information onto the Graphics Formatting block for display on the application LCD.

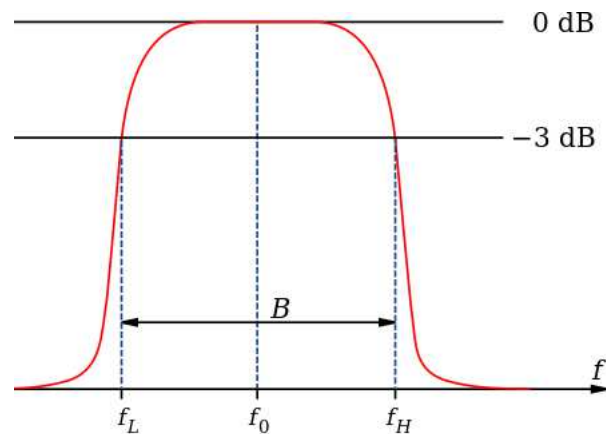
Graphic Equalizers

As a black box, a graphic equalizer has left/right inputs and left/right outputs:



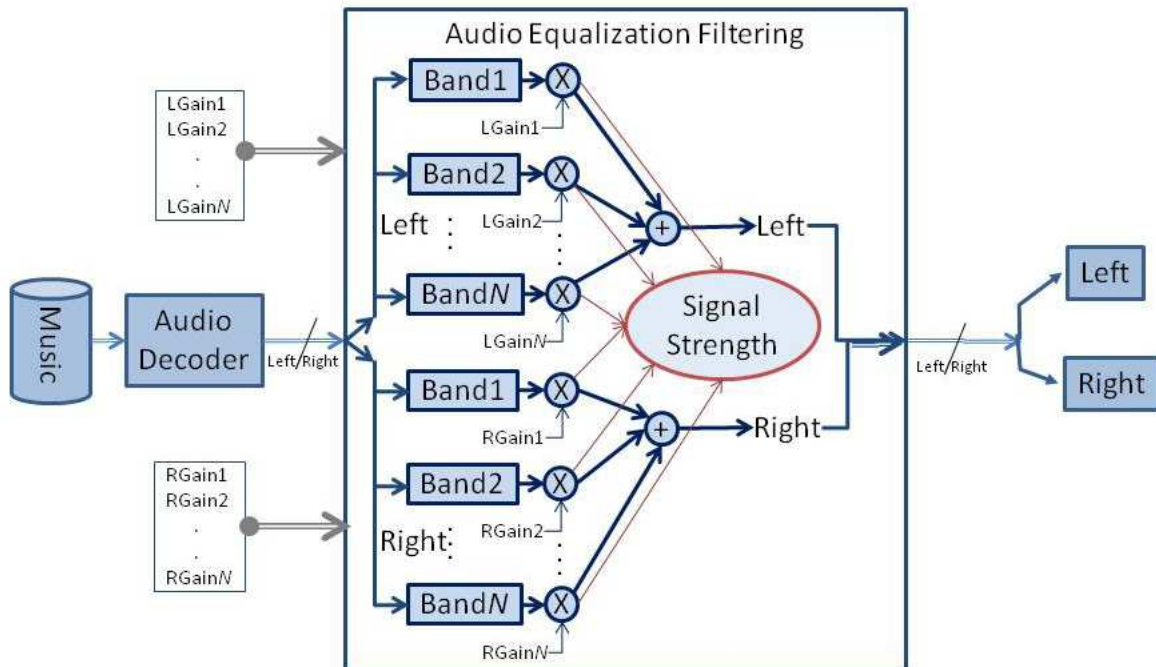
In this example, frequency bands are centered at 50 Hz, 150 Hz, 300 Hz, ... , 10 KHz, and 12.5 KHz. Signal strength is plotted for each frequency band by a stack of green bars, more bars meaning a stronger signal. Below each stack of green bars is a slider that is used to adjust the filter gain for each band, which is centered at the frequency shown. Moving the red tab up increases filter gain, increasing signal strength. Moving the red tab down decreases filter gain, decreasing signal strength.

A typical band filter passes signals centered at a frequency (f_0) and attenuates signals outside of a pass band (f_L to f_H). The pass band gain, shown below as 0 dB, or unity gain, can be adjusted up or down to increase or decrease signal strength in the band (f_L to f_H).



Graphic equalizer filters adjust the spectral content of music by filtering left and right stereo signals through a bank of parallel filters, summing the results to create the output left and right signals. Each filter passes part of the signal's spectrum. Added together again after filtering the left/right signals are reconstructed with modified spectral content. Increasing the gain of a filter will emphasize the signal in that filter's frequency band. Decreasing the gain of a filter will de-emphasize the signal in that filter's frequency band.

A typical graphic equalizer filter bank can be represented by:



The Audio Decoder takes raw binary music and decodes it into a stream of 16 or 24 bit integers, with a pair of such integers representing a single left/right sample of music. Typically these samples are played at 44,100 or 48,000 sample per second. Signals from the left or right channel are fed into a bank of parallel band filters. The output of each filter is multiplied by a user-adjustable gain and then summed together to create a left or right output signal, which is then sent via a DAC to speakers. Each filter output is used to update the signal strength of each filter's output. It is this data that is displayed on the Graphic Equalizer screen.

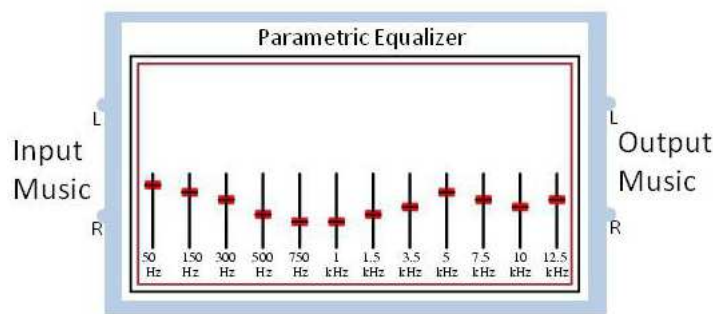
Filters for each band are designed to overlap across the signal's spectrum. The signal's spectrum can be divided into equally sized band or into bands that increase by a factor of two with each higher frequency band.

```
% Linear spacing: 6 bands from Dc to FS/2, with FS = 44,100 Hz
FcFilters = [ 0 4410 8820 13230 17640 22050 ];
```

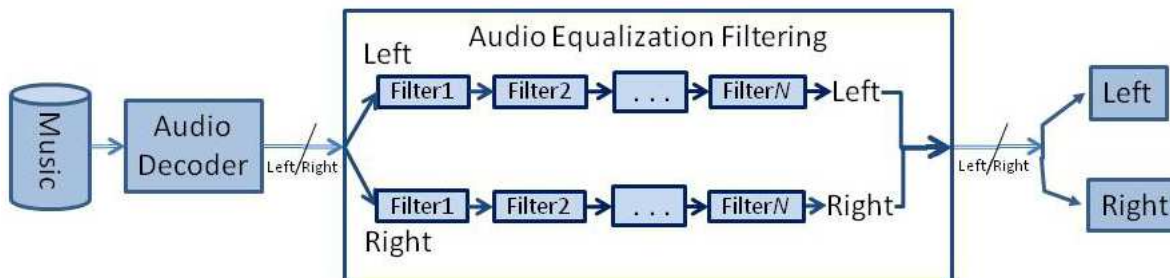
```
% Octave spacing of band centers:
FcFilters = [ 125/2 125 250 500 1000 2000 4000 8000 16000 ];
```

Parametric Equalizers

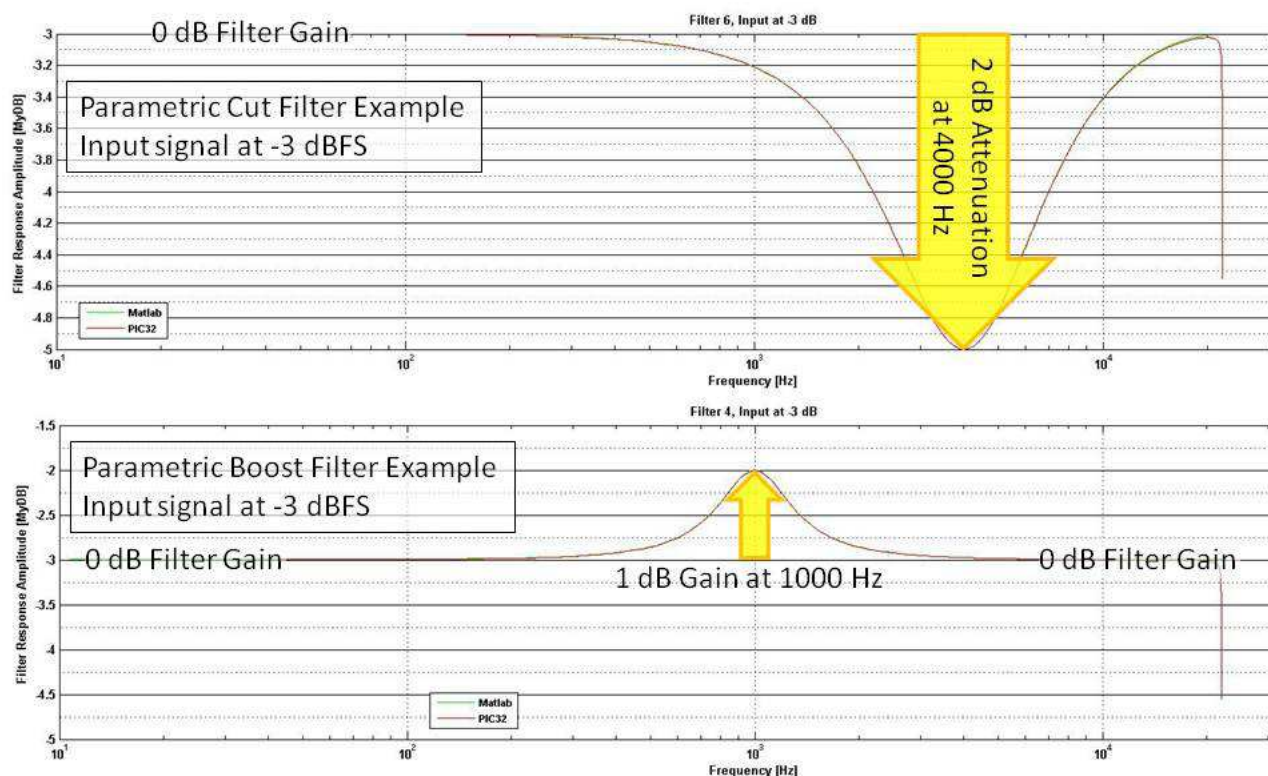
Just like a graphic equalizer, as a black box a parametric equalizer has left/right inputs and left/right outputs. But in most cases the display of signal strength by frequency band is missing.



Examining the filtering structure of a parametric equalizer reveals why the signal strength display is missing. As seen below, parametric equalization is accomplished by a chain of back-to-back filters, with each filter having 0 dB (unity) gain outside of specified frequency range.



Here are two parametric equalization filter examples:



Since the filters are in a back-to-back chain rather than in parallel, there is no way to directly measure the signal strength (energy) of a particular band using a filter's output. Of course signal strength (energy) can be measured using a Fast Fourier Transform (FFT) but such a calculation is very expensive when filter bands are not equally spaced. Another disadvantage of parametric filters is that, unlike a graphic equalization filter, there is no easy way to adjust the boost gain or cut attenuation of each filter. You essentially have to design a family of parametric filters, with a range of gains, and keep them in a look-up table. With graphic equalization filtering all you need do is adjust the gain multiplier after each filter to change the filter's gain.

On the other hand, parametric equalization filters can accomplish an overall gain adjustment across the signal's spectrum with fewer filters. Thus for fixed equalization, such as speaker correction, that don't need real-time gain adjustments, parametric equalization filtering is the preferred approach.

1.2 Library Overview

In music playback adjustments can be made to the music before digital bits are converted into analog voltages that are played by speakers. These adjustments can correct deficits in the recorded music, mitigate problems in the speakers being used, or simply correct for the room's acoustics - all focused on improving the user's experience of the music.

This [library](#) supports filtering in two bit widths:

- 16 bits (Q15) provides maximum computational efficiency but with reduced filtering accuracy. Careful attention must be paid to input signal levels and filter gains to avoid overflow and truncation.
- 32 bits (Q31) provides greater accuracy. It is necessary for 24-bit input signals. 32 bits provides 8 bits (48 dB) headroom for 24 bit input signals and 16 bits (96 dB) headroom for 16 bit input signals.

Filter examples are provided and filter design tools for Matlab(tm)/Octave are provided to aid in the creation of new filters.

Actual filter performance on PIC32 devices can be measured using validation tool projects and Matlab(tm)/Octave processing scripts provided with the [library](#).

Graphic Equalizers

A traditional graphic equalizer slices sound into several frequency bands, filters each band, and then reassembles the output from each band into the final output signal. Typically there are two banks of band filters, one for the left stereo signal and another for the right stereo signal. The filter for each frequency band attenuates (stops) sound energy outside of some frequency range while providing an overall gain for the frequencies within the filter's passband. Adjusting each band's gain allows the user to adjust the overall frequency response of the system.

Here is a crude, yet still informative, diagram of a 6-band graphic equalizer:

```

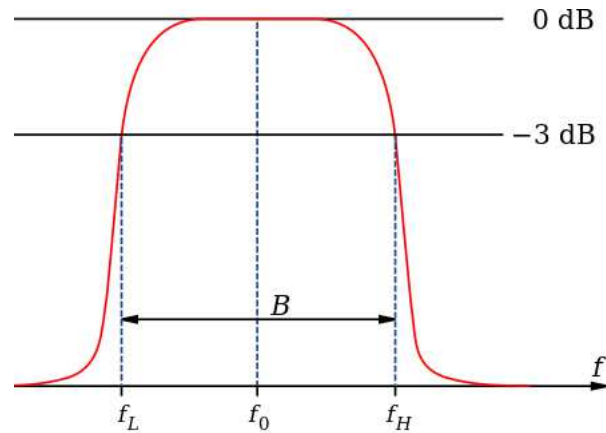
XinLeft ---+-->Filter[1]-->(x)----+
          |   LeftGain[1]-^      V
          +-->Filter[2]-->(x)--->(+)--+
          |   LeftGain[2]-^      V
          +-->Filter[3]-->(x)----->(+)--+
          |   LeftGain[3]-^      V
          +-->Filter[4]-->(x)----->(+)--+
          |   LeftGain[4]-^      V
          +-->Filter[5]-->(x)----->(+)--+
          |   LeftGain[5]-^      V
          +-->Filter[6]-->(x)----->(+)--> YoutLeft
          LeftGain[8]-^

XinRight---+-->Filter[1]-->(x)----+
          |   RightGain[1]-^     V
          +-->Filter[2]-->(x)--->(+)--+
          |   RightGain[2]-^     V
          +-->Filter[3]-->(x)----->(+)--+
          |   RightGain[3]-^     V
          +-->Filter[4]-->(x)----->(+)--+
          |   RightGain[4]-^     V
          +-->Filter[5]-->(x)----->(+)--+
          |   RightGain[5]-^     V
          +-->Filter[6]-->(x)----->(+)--> YoutRight
          RightGain[8]-^

```

The advantage of this approach is that the filters (filter coefficients) don't change, only the gain adjustment multiplication factor changes.

Here's the filter response of a typical bandpass filter:



Note that outside of the filter's passband $[f_L, f_H]$ the signal is attenuated. Here's a plot showing the response of a four-band graphic equalizer, with the overall response shown as a dashed red line.



Parametric Equalizers

Parametric filters have a different structure. Instead of separate band filters operating in parallel, a cascade of back-to-back filters is applied to left and right channels:

```

XinLeft -->Filter[0]-->Filter[1]-->Filter[2]-->Filter[3]--+
|
+-----+
|
+-->Filter[4]-->Filter[5]-->Filter[6]-->Filter[7]-->(x)--> YoutLeft
                                   LeftGain[7]-^

XinRight -->Filter[0]-->Filter[1]-->Filter[2]-->Filter[3]--+
|
+-----+
|
+-->Filter[4]-->Filter[5]-->Filter[6]-->Filter[7]-->(x)--> YoutRight
                                   RightGain[7]-^

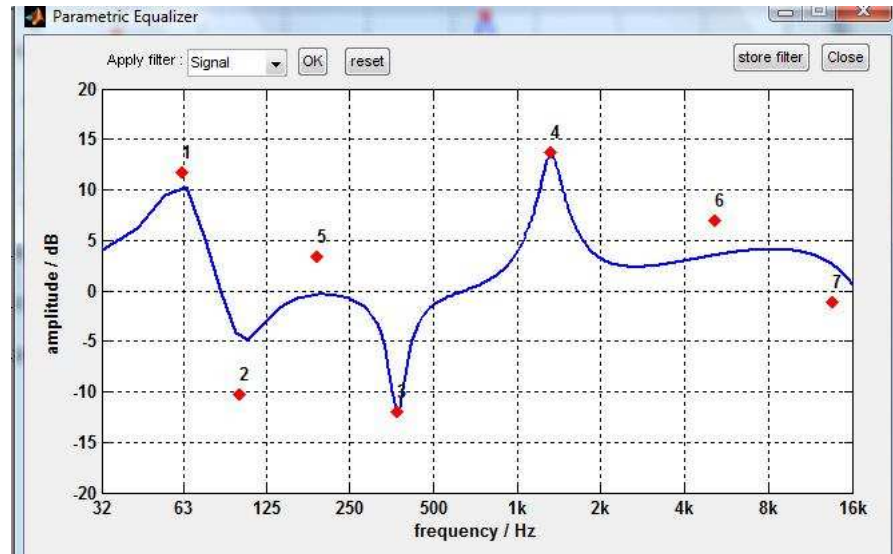
```

Here the same filters are applied to both left and right channels, but channel-specific filters can be used as well.

Since the filters are cascaded, each filter must pass all frequencies while making an adjustment to the signal at a particular

frequency, either providing gain to increase the signal or attenuation to reduce the signal. A cascade of filters adjusts the signal's frequency response at a set of frequencies. Starting out, you can think of a cascade of all-pass filters, each filter doing nothing except passing the signal without any gain or attenuation. The overall signal response would then be a flat response of zero dB from DC to the maximum frequency, like a stretched rubber band at zero dB.

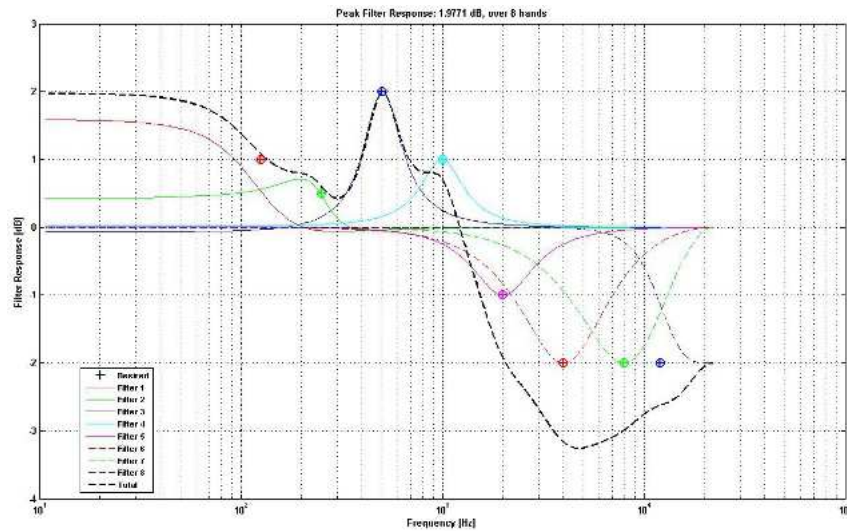
The flat response can then be adjusted, like pulling on the stretched rubber band at various places, to increase or to decrease the signal at particular points. Here is the GUI for a MATLAB (tm) tool that designs parametric filters. Note how the signal is manipulated at seven distinct locations in the frequency band.



Microchip provides a parametric equalization filter design tool that works on both Matlab(tm) and Octave. It has a simpler, text-based control interface:

The screenshot shows the 'Parametric Equalization Filter Design' dialog box. It contains several input fields and tables for configuring the filter. The 'Filter_Bit_Width' is set to 32. The 'Sample_Rate [Hz]' is set to 44100. The 'Center_Frequencies [Hz]' table has values: 125, 250, 500, 1000, 2000, 4000, 8000, 12000. The 'Gains [dB]' table has values: 1, 0.5, 2, 1, -1, -2, -2, -2. The 'Bandwidths [Hz]' table has values: NA, 176.777, 353.553, 707.107, 1414.21, 2828.43, 5656.85, NA. The 'Shelving_Qs [Low,High]' table has values: 0.707107, NA, NA, NA, NA, NA, NA, 0.707107. The 'Plot_Command [plot,semilogx]' field is set to 'semilogx'. The dialog includes 'OK' and 'Cancel' buttons.

This design dialog produced this filter:



The advantage of cascading filters is that the filters don't need to waste effort attenuating out-of-band frequency but instead just adjust the frequency response by adding gain to increase the signal or attenuation to decrease the signal. The disadvantage is that the size of each filter's manipulation must be known to design the filter's coefficients. So it is not possible to adjust the cascade's overall frequency response once the coefficients are loaded into firmware.

Of course you could always design a family of filters, with a range of gains/attenuations, and store them in a lookup table. But that would be far more work than simply adjusting a gain factor, as was seen in the case of the graphic equalizer.

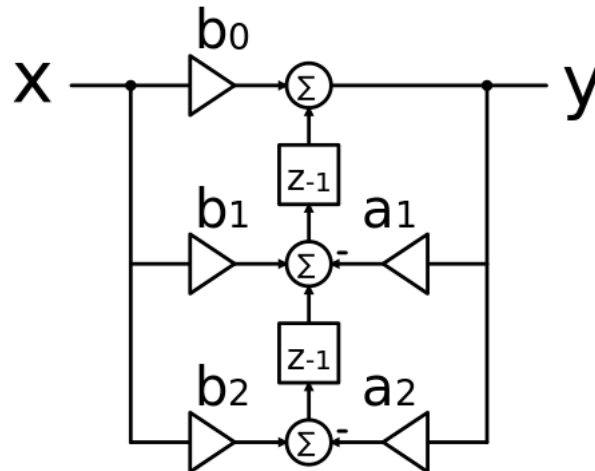
Equalizer Filter Implementation

Up to this point we have treated each filter as a black box. Now we examine what's inside of each filter. For music it is the signal's amplitude that carries its information, phase is unimportant. This allows the use of **Infinite Impulse Response (IIR)** filters for equalization filtering rather than the more computationally expensive **Finite Impulse Response (FIR)** filters. IIR filters don't conserve phase, but phase is unimportant.

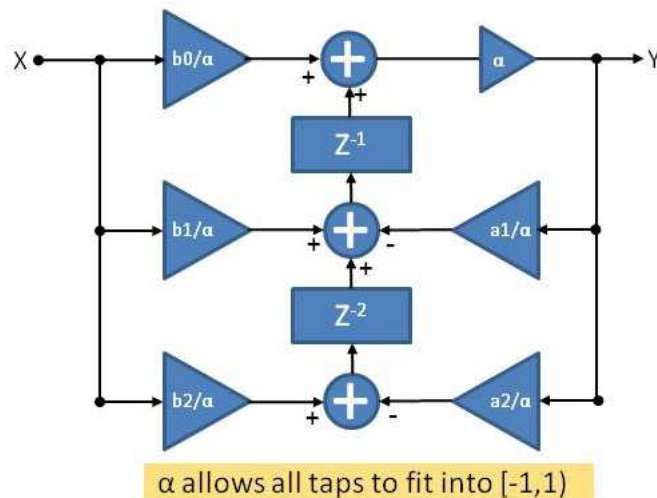
The simplest IIR filter is called a BiQuad, because it has a bi-quadratic transfer function when viewed in the Z domain. The equation for a biquad IIR filter is:

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) - a_1y(n-1) - a_2y(n-2)$$

This shows how to calculate the latest output, $y(n)$, using the past three inputs and past two outputs. The best way to calculate $y(n)$ is the "Transposed Direct Form II", which only needs two memory slots, shown as Z^{-1} blocks below:



In the [library](#)'s filtering functions the gain stages ($>$ and $<$) are multiplies and the Z^{-1} blocks are integer variables. All the mathematics in the filtering primitives (the adding and multiplying) is done in "fixed point" arithmetic, either Q0.15 for 16 bit inputs or Q0.31 for 32 bit inputs. This implies that the coefficients (the a 's and the b 's) must fit between $-1 \leq \text{value} < +1$. For a_1 , b_0 , and b_2 , this is no problem, but typically a_1 and b_1 don't fit into this range. In fact $-2 < a_1, b_1 < +2$. Thus we have to divide all the coefficients by an *alpha* factor to get them to fit in the range $-1 \leq \text{value} < +1$. We then have to put the *alpha* factor back in before outputting y .



In all cases $\alpha = 2$ is all you need. This is implemented in software by setting $\log_2 \text{Alpha} = 1$.

So to prepare a filter to use by the equalization [library](#) divide all coefficients by 2 and set $\log_2 \text{Alpha} = 1$.

Specifying Filters

Filters are specified using the [EQUALIZER_FILTER](#) or [EQUALIZER_FILTER_32](#) typedef to define an array of structures. Since filter definition `.h` files are automatically generated by the Matlab(tm)/Octave filter design scripts, the exact details of the filter structure are of little importance. The only useful modification of these files is to clone a filter set into left and right filters for use in stereo music filtering.

1.3 Resource Requirements

Resource Requirements

The data memory and program flash needed for filtering are so small for most audio applications as to be inconsequential. For example, the assembly routine [AUDIO_EQUALIZER_Cascade8inQ31](#) uses only 40 words of flash memory. The definition of single 16 bit biquad IIR filter only needs seven 32-bit words of RAM, while the equivalent 32-bit IIR biquad filter definition needs just ten 32-bit words. So for a 8 band graphic equalizer filter structure, with two IIRs/band, needs only $7 \times 8 \times 2 = 112$ 32-bit words for filter memory. The equivalent 32-bit structure needs $10 \times 8 \times 2 = 160$ words.

However, the processing required to execute the filters on each new left/right stereo sample can easily account for over 50% of the processor's bandwidth. The [Filtering Performance](#) section below provides the benchmarks needed to estimate the processing load for any filter architecture and data rate.

Here is a summary table of the millions of instructions per second (MIPS) required for various Graphic Equalizer Equalization filters, assuming stereo (left/right) data at 44.1 KSPS or 48 KSPS:

Graphic Equalizer Filtering MIPS Requirements					
To Filter Stereo Music Samples at Given Sample Rate					
# Bands	# IIRs/Band	MIPS Requirements			
		Without Display		With Display	
		44.1 KSPS	48 KSPS	44.1 KSPS	48 KSPS
4	2	27.43	29.86	29.90	32.54
5	2	34.05	37.06	37.04	40.32
6	2	40.66	44.26	44.01	47.90
7	2	47.36	51.55	51.24	55.78
8	2	53.89	58.66	58.39	63.55
With Display: Measuring signal strength per frequency band					
Without Display: No signal strength measurements					
Measured on PIC32MX450F256L processor using 16 bit Filters					
Typical Applications					

For parametric equalization filters, the following table shows the MIPS required for various filter chain lengths:

Parametric Equalizer Filtering MIPS Requirements					
To Filter Stereo Music Samples at Given Sample Rate					
	MIPS Requirements				
	16 Bit Filters		32 Bit Filters		
	16 Bit Data		24-32 Bit Data		
#Filters	44.1 KSPS	48 KSPS	44.1 KSPS	48 KSPS	
2	6.8796	7.488	8.1144	8.832	
3	9.3492	10.176	11.2014	12.192	
4	11.8188	12.864	14.2884	15.552	
5	14.2884	15.552	17.3754	18.912	
6	16.758	18.24	20.4624	22.272	
7	19.2276	20.928	23.5494	25.632	
8	21.6972	23.616	26.6364	28.992	
9	24.1668	26.304	29.7234	32.352	
10	26.6364	28.992	32.8104	35.712	
11	29.106	31.68	35.8974	39.072	
12	31.5756	34.368	38.9844	42.432	
13	34.0452	37.056	42.0714	45.792	
14	36.5148	39.744	45.1584	49.152	
15	38.9844	42.432	48.2454	52.512	
Measured on PIC32MX450F256L processor					
Typical Applications					

Note that the application may not support full filter processing for the desired number of filter bands during debugging. An easy workaround is to use a less computationally expensive set of filters (i.e. fewer bands) while debugging. After debugging the full set of filters can be applied when the application is optimized for speed and size ($O = s$ or $O = 3$ in the compiler).

1.4 Glossary of Terms

Frequently Used but Possibly Obscure Terms:

Band - A part of a signal's frequency spectrum defined by a distinct upper and lower frequency limits, or a center frequency and bandwidth.

Band Energy - signal strength or energy of a given band, typically measured at the output of the band filter.

Band Energy Units - signal strength or band energy is reported as a voltage squared or absolute voltage. It can be reported in volts or in dB.

- BAND_ENERGY_RMS_VOLTS - - root mean squared voltage, with value of 1 representing the maximum possible signal
- BAND_ENERGY_RMS_DBFS - in dB re Full Scale using RMS energy estimate
- BAND_ENERGY_PSEUDORMS_VOLTS, - Pseudo RMS using absolute value instead of voltage squared
- BAND_ENERGY_PSEUDORMS_DBFS - in dB re Full Scale using Pseudo RMS energy estimate

Pseudo RMS - Energy estimated by sum of absolute values instead of voltage squared. The average value is adjusted so that pseudo RMS of sine wave is same value as the RMS of the same sine wave.

1.5 Release Notes

Audio Equalizer Filtering Library Version:

0.1Beta **Release Date:** 18 November 2013

This is the first release of the [library](#). The interface can change in the beta and/or 1.0 release.

1.6 SW License Agreement

(c) 2013 Microchip Technology Inc.

Microchip licenses this software to you solely for use with Microchip products. The software is owned by Microchip and its licensors, and is protected under applicable copyright laws. All rights reserved.

SOFTWARE IS PROVIDED "AS IS" MICROCHIP EXPRESSLY DISCLAIMS ANY WARRANTY OF ANY KIND, WHETHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL MICROCHIP BE LIABLE FOR ANY INCIDENTAL, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, HARM TO YOUR EQUIPMENT, COST OF PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY OR SERVICES, ANY CLAIMS BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF), ANY CLAIMS FOR INDEMNITY OR CONTRIBUTION, OR OTHER SIMILAR COSTS.

To the fullest extent allowed by law, Microchip and its licensors liability shall not exceed the amount of fees, if any, that you have paid directly to Microchip to use this software.

MICROCHIP PROVIDES THIS SOFTWARE CONDITIONALLY UPON YOUR ACCEPTANCE OF THESE TERMS.

1.7 Using the Library

This section describes the basic architecture of the Audio Equalizer Library and provides information and examples on how to use it.

Interface Header File: framework/math/[audio_equalizer.h](#)

The interface to the Audio Equalizer Library is defined in the "framework/math/[audio_equalizer.h](#)" header file.

1.7.1 Configuring the Library

AUDIO_EQUALIZER_MAX_NBANDS

There is one configuration #define's in the [library](#): [AUDIO_EQUALIZER_MAX_NBANDS](#):

```
#define AUDIO_EQUALIZER_MAX_NBANDS 16
```

defines the maximum number of energy bands that the [library](#) can support. For example, if there are 8 frequency bands in the graphical equalizer then the number of frequency bands is $2 * 8 = 16$, since left and right stereo channels have separate filters and energy estimates for each band.

The index of energy bands between left and right channels is up to the user. { Left0, Right0, Left1, Right1, ..., LeftN, RightN} or {Left0, Left1, ... , LeftN, Right0, Right1, ..., RightN} ordering of bands will both work.

1.7.2 Fixed Point Data and Mathematics

C Language Native Data Types

The C programming language supports integer and floating point data types. It does not support fractional data types. Thus you can only represent $1/2$ as a floating point constant (0.5) but that requires using floating point mathematics. For many embedded applications floating point mathematics is too slow and needs too much memory to be of practical use.

Fixed Point Addition and Subtraction

Fixed point ($Q_m.n$ or Q_n) data types and associated mathematics support fractional data without using floating point. It allows the use of integer data types instead of floating point. The key idea is to think of fractional data as a pair of numbers, the numerator and denominator. A signed 16 bit integer ranges from -32768 to +32767. So using 16 bits you can represent $1/2$ as $16384/32768$. Thus $1/2$ is represented by M/N , where $M = 16384$ and $N = 32768$.

Now the trick comes that we just do integer math using M and keep N hidden (in our heads). Each fixed point data type has an implied numerator (M). For example the $Q0.15$ 16 bit fixed point data type has $N = 32768$.

Adding fixed point fractions simply means adding the numerators:

$$1/2 + 1/4 = 16384/32768 + 8192/32768 = (16384 + 8192)/32768 = 24576/32768 = 3/4$$

So in C you simply add the numerators and keep the denominator (32768) in your head:

$$1/2 + 1/4 = 16384 + 8192 = 24576$$

Fixed Point Data Types

Fixed point data types are defined as typedefs in the Audio Equalizer Filtering [library](#) file [audio_equalizer_fixedpoint.h](#).

Each data type is of the form Qm.n, where m is the number of integer bits and n is the number of fractional bits. Thus [libq_q15d16_t](#) is a 32 bit format with 1 one sign bit, 15 integer bits, and 16 fractional bits:

```

-3-----2-----1-----
10987654321098765432109876543210
-----
Siiiiiiiiiiiiiiiiiffffffffffffffffff

```

With 16 fractional bits the implied denominator $N = 65536$:

```

float Xfloat;
libq_q15d16_t Xq15d31;

Xfloat = Xq15d31/65536.0;

```

The most common data types are the 16-bit [libq_q0d15_t](#) (or [libq_q15_t](#)):

```

-----1-----
5432109876543210
-----
Sfffffffffffffffff

```

As mentioned above, Q0.15 (or Q15) has an implied $N = 32768$:

```

float Xfloat;
libq_q15_t Xq15;

Xfloat = Xq15/32768.0;

```

And the 32-bit [libq_q0d31_t](#) (or [libq_q31_t](#)):

```

-3-----2-----1-----
10987654321098765432109876543210
-----
Sfffffffffffffffffffffffffffffffff

```

Q0.31 (or Q31) has an implied $N = 2^{31}$:

```

float Xfloat;
libq_q31_t Xq31;
Xfloat = Xq31/((float)2<<31);

```

Fixed Point Multiplication and Division

Fixed point addition and subtraction is very easy since we just add or subtract numerators. But multiplication and division need some tweaks to convert integer multiplication and division into fixed point multiplication and division. (PIC32 assembly has instructions that do fixed point multiplication/division correctly.) Let's look at multiplying $1/2 * 1/4 = 1/8$:

```

1/2 * 1/4 = 16384/32768 * 8192/32768 = (16384 * 8192)/(32768*32768) =
134,217,728/1,073,743,824 = 1/8

```

Ignoring the denominators, and just multiplying the numerators we have:

```

1/2 * 1/4 = 16384 * 8192 = 16384 * 8192 = 134,217,728

```

So the first thing you notice is multiplying two 16-bit integers produces a 32-bit integer ($16+16 = 32$).

So to convert it back into a 16 bit integer, just shift by 16 bits:

```

1/2 * 1/4 = 16384 * 8192 = 16384 * 8192 = 134,217,728>>16 = 2048

```

But

$$1/8 = 4096/32768$$

So we are off by a factor of two. Thus you have the rule that you must left shift integer multiplication by one bit to produce fractional multiplication. Similarly integer division must be *right* shifted one bit to produce fractional division.

Here's a code snippet that multiplies two Q0d15 numbers:

```
libq_q15_t A16, B16, C16;
libq_q31_t Temp32;

Temp32 = A16 * B16;
C16     = Temp32>>(16-1);
```

This example can be simplified since 32 bit integers are assigned to 16 bit by copying the lower 16 bits:

```
C16 = (A16 * B16)>>(16-1);
```

Note that this works because C promotes all 16-bit multiplication into 32 bit. But for 32 bit multiplication you have to explicitly cast one of the multiplicands into a 64 bit integer:

```
libq_q31_t A32, B32, C32;
libq_q63_t Temp64;

Temp64 = A32 * (libq_q63_t)B32;
C32     = Temp64>>(32-1);
```

Or simply:

```
C32 = (A32 * (lib1_q63_t)B32)>>(32-1);
```

1.7.3 Core Exception Handling

Fixed Point Overflow

All mathematics in the [library](#) is "fixed point", in which an integer variable is used to represent fractional values without resorting to floating point. For example, the Q0.15 fixed point type uses a 16-bit signed integer to represent fractional values between -1 and +1.

If signals are too large in amplitude or filters badly designed the assembly filtering routines can produce overflow core exceptions. Out of the box the "weak" exception handler installed as part of the compiler simply dumps the application into a while(1) loop when any core exception occurs. Thus any application using the default exception handler would simply stop working whenever an overflow occurs.

Dedicated Exception Handler

Instead the filtering application should continue to work, even if it produces badly filtered output. The snaps, pops, and noise produced when overflows occur will alert the user that something is amiss. Then the user can reduce band gain until the filters are not over driven and thus stop the snaps/pops/noise. To support this behavior an exception handler tailored for equalization filtering must be used instead of the compiler's default.

The files `audio_eq_exception-handler.c` and `audio_eq_general-exception.S` must be include in the application's MPLAB.X project. The assembly (.S) files provides additional support for saving and restoring processor registers during exceptions. The .c file supports recovery from overflow exceptions that allow filtering (and the application) to continue.

1.7.4 Filtering Performance

Benchmark Results

Filtering performance was measured using a PIC32MX450F256L processor, with the C test fixture at optimization level zero. The tables below show the instruction count for filtering a single input sample to produce an output sample. These results are for filtering alone, without any band energy estimates.

The column "All Filters" shows the instruction count for a single invocation of the filtering primitive. Some primitives can execute more than one IIR biquad filter, so the "Single Filter" column shows the average cost per IIR biquad for each primitive. (This is the "All Filters" column divided by the number of IIRs executed with each call.)

Q15 Filtering Primitives:

Audio Equalizer Function	M4K Single Sample Instruction Count			
	N	M	All Filters	Single Filter
AUDIO_EQUALIZER_IIRinQ15andC*			46	1
AUDIO_EQUALIZER_IIRinQ15andC**			56	1
AUDIO_EQUALIZER_IIRinQ15FastC			98	1
AUDIO_EQUALIZER_IIRinQ15			34	1
AUDIO_EQUALIZER_Cascade2inQ15			78	2
AUDIO_EQUALIZER_Cascade6inQ16			246	8
AUDIO_EQUALIZER_Parallel4x2inQ15			288	8
AUDIO_EQUALIZER_Parallel8x2inQ15			550	16
AUDIO_EQUALIZER_ParallelNx2inQ15	4		306	8
	5		378	10
	6		448	12
	7		518	14
	8		586	16
	9		658	18
AUDIO_EQUALIZER_ParallelNxMinQ15	4	2	316	8
	5	2	384	10
	5	3	530	15
	6	2	454	12
	8	2	596	16
	9	4	1186	36

* bApplGain = false, Optimization = 3

** bApplGain = true, Optimization = 3

Q31 Filtering Primitives:

Audio Equalizer Function	M4K Single Sample Instruction Count			
	N	M	All Filters	Single Filter
AUDIO_EQUALIZER_IIRinQ31andC*			60	1
AUDIO_EQUALIZER_IIRinQ31andC**			78	1
AUDIO_EQUALIZER_IIRinQ31			44	1
AUDIO_EQUALIZER_Cascade2inQ31			92	2
AUDIO_EQUALIZER_Cascade6inQ31			302	8
AUDIO_EQUALIZER_Parallel4x2inQ31			342	8
AUDIO_EQUALIZER_Parallel8x2inQ31			672	16
AUDIO_EQUALIZER_ParallelNx2inQ31	4		374	8
	5		456	10
	6		542	12
	7		626	14
	8		714	16
	9		790	18
AUDIO_EQUALIZER_ParallelNxMinQ31	4	2	382	8
	5	2	466	10
	5	3	646	15
	6	2	550	12
	8	2	722	16
	9	4	1404	36

* bApplGain = false, Optimization = 3

** bApplGain = true, Optimization = 3

Q15 versus Q31 Performance:

Filtering in 32 bits ranges from 39% to 18% more expensive:

Comparison of 16-Bit versus 32-Bit Filtering Performance		M4K Single Sample Instruction Count		
16-Bit Audio Equalizer Function	32-Bit Audio Equalizer Function	Q15	Q31	Q31/Q15
AUDIO_EQUALIZER_IIRinQ15andC*	AUDIO_EQUALIZER_IIRinQ31andC*	46	60	130%
AUDIO_EQUALIZER_IIRinQ15andC**	AUDIO_EQUALIZER_IIRinQ31andC**	56	78	139%
AUDIO_EQUALIZER_IIRinQ15	AUDIO_EQUALIZER_IIRinQ31	34	44	129%
AUDIO_EQUALIZER_Cascade2inQ15	AUDIO_EQUALIZER_Cascade2inQ31	78	92	118%
AUDIO_EQUALIZER_Cascade8inQ16	AUDIO_EQUALIZER_Cascade8inQ31	246	302	123%
AUDIO_EQUALIZER_Parallel4x2inQ15	AUDIO_EQUALIZER_Parallel4x2inQ31	288	342	119%
AUDIO_EQUALIZER_Parallel8x2inQ15	AUDIO_EQUALIZER_Parallel8x2inQ31	550	672	122%
AUDIO_EQUALIZER_ParallelNx2inQ15	AUDIO_EQUALIZER_ParallelNx2inQ31	306	374	122%
		378	456	121%
		448	542	121%
		518	626	121%
		586	714	122%
AUDIO_EQUALIZER_ParallelNxMinQ15	AUDIO_EQUALIZER_ParallelNxMinQ31	658	790	120%
		316	382	121%
		384	466	121%
		530	646	122%
		454	550	121%
		596	722	121%
		1186	1404	118%

* bApplGain = false, Optimization = 3

** bApplGain = true, Optimization = 3

Average: 121%

Filtering for a Graphical Equalizer

If band energy estimates are added to the filtering, as shown in the documentation for [AUDIO_EQUALIZER_BandEnergyNSamplesSet](#), the following results are measured:

Instruction count to filter (Left,Right) Samples						
		Instruction Count				
# Bands	# IIRs/Band	Without Display		With Display		Display Surcharge
		Measured	Per IIR	Measured	Per IIR	
4	2	622	38.8750	678	42.3750	9.0%
5	2	772	38.6000	840	42.0000	8.8%
6	2	922	38.4167	998	41.5833	8.2%
7	2	1074	38.3571	1162	41.5000	8.2%
8	2	1222	38.1875	1324	41.3750	8.3%
With Display: Measuring signal strength per frequency band						
Without Display: No signal strength measurements						
Measured on PIC32MX450F256L processor using 16 bit Filters						

The instruction counts shown are for filtering both channels (left/right) in a stereo signal. These numbers are smaller than twice the instruction counts shown above because the final gain adjustment built into all assembly primitives was not included in the filtering example tested.

The "Display Surcharge" column shows the additional processing required over just filtering to use every output sample for band signal strength (energy) measurements.

Estimating Processing Requirements - An Example

Assume that stereo music is decoded at a rate of 44.1 KSPS. Then a 6x2 graphic equalizer with energy estimates will need 998

instructions for each left/right music sample to produce an output. Left/right music sample arrive at a rate of 44100 samples per second. So the processing bandwidth required to keep up is given by

Processing Bandwidth = 1 left/right samples * 44100 samples/second * 998 instructions/sample = 44,011,800 instructions/second = 44.01 MIPS

Benchmark Methodology

Benchmarking firmware was run on a PIC32MX450F256L. Here are code snippets that show how benchmarking measurements were made:

```
uint16_t timerStart, timerEnd, timerOverhead, testCycles;

// Start timer
asm volatile("mtc0    $0,$9");
asm volatile("mfc0    %0, $9" : "=r"(timerStart));

FilterInput(XinQ15,XinQ15,&YoutLeft,&YoutRight);

//Stop timer, determine elapsed time.
asm volatile("mfc0    %0, $9" : "=r"(timerEnd));
testCycles = 2*(timerEnd - timerStart); // eval cycles for function under test
```

Multiple measurements were made, typically over 256 samples. Also, the overhead of simply starting and stopping the timer was measured by replacing the FilterInput call with blocks of asm("NOP")'s :

```
// Measure timer overhead
asm volatile("mtc0    $0,$9"); // Start timer
asm volatile("mfc0    %0, $9" : "=r"(timerStart));

asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");//5
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");//10
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");//20
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");//30
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");//40
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");//50

asm volatile("mfc0    %0, $9" : "=r"(timerEnd)); //Stop timer, determine elapsed time.
timerOverhead = 2*(timerEnd - timerStart) - 50; //Calculate overhead
```

1.7.5 Application Examples

Updating a Graphic Equalizer Display

```
#include "audio_equalizer.h"
#include "audio_equalizer_fixedpoint.h"

#include "myStereoFilters6x2_Q15.h"
#define NBANDS 6
#define NFILTERS 2

uint16_t iBand;
libq_q0d15_t XinLeft,XinRight,YoutBand, YoutLeft, YoutRight;
libq_q0d15_t bandEnergyDBFS;

// Define labels
GRAPHIC_EQUALIZER_LabelsSet("Left","Right",(void*)LARGE_FONT);
```



```

// Signal strength will be measured in RMS Volts in dB re Full scale, -30 dBFS to 0 dBFS
GRAPHIC_EQUALIZER_DisplayScaleSet (GFX_EQUAL_SIGNAL_STRENGTH,-30<<16,0<<16);

// Filter gains will range from -10 dB to +10 dB
GRAPHIC_EQUALIZER_DisplayScaleSet (GFX_EQUAL_FILTER_GAIN,-10<<16,10<<16);

// Draw graphic equalizer display
GRAPHIC_EQUALIZER_Create( 8, 6, //Xleft,Ytop
                        8, 4, //BarWidth, BarHeight
                        8,16); //nBands, nBars

// Setup to measure signal energy in dB re Full Scale (dBFS)
AUDIO_EQUALIZER_BandEnergySumsInit (2*NBANDS,BAND_ENERGY_RMS_DBFS);

while ( 1 )
{
    if ( bGotInput() )
    {
        // Get XinLeft,XinRight

        // Execute equalizer filtering and signal strength updates.
        // See Audio Equalizer Filtering Library for example code to implement this.
        FilterInputPlusEnergy(XinLeft,XinRight,&YoutLeft,&YoutRight);

        // Send YoutRight,YoutRight
    } //end if ( bGotInput )

    if ( bUpdateDisplay() )
    {
        // Memory update for each frequency band, left and right channels
        for ( iBand = 0; iBand < NBANDS; iBand++ )
        {
            // Left channel: Update signal strength
            bandEnergyDBFS = AUDIO_EQUALIZER_BandEnergyGetQ15(iBand,true);
            GRAPHIC_EQUALIZER_BandValueUpdate (GFX_EQUAL_SIGNAL_STRENGTH,GFX_EQUAL_CHANNEL_LEFT,iBand,bandEnergyDBFS);

            // Right Channel: Update signal strength
            bandEnergyDBFS = AUDIO_EQUALIZER_BandEnergyGetQ15(iBand+NBANDS,true);
            GRAPHIC_EQUALIZER_BandValueUpdate (GFX_EQUAL_SIGNAL_STRENGTH,GFX_EQUAL_CHANNEL_RIGHT,iBand,bandEnergyDBFS);

        } //end for ( iBand = 0; iBand < NBANDS; iBand++ )

        // Refresh entire display at same time
        GRAPHIC_EQUALIZER_BandDisplayRefresh(GFX_EQUAL_CHANNEL_LEFT, -1); // Refresh all
bands for Left
        GRAPHIC_EQUALIZER_BandDisplayRefresh(GFX_EQUAL_CHANNEL_RIGHT,-1); // Refresh all
bands for Right

    } //end if ( bUpdateDisplay )

} //end while ( 1 )

```

Filtering and Measuring Signal Strength (Energy)

```

#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>

#include "math/audio_equalizer/audio_equalizer.h"
extern uint16_t AUDIO_EQUALIZER_nSamples;

```

```

extern libq_q15d15_t AUDIO_EQUALIZER_BandEnergySumQ15[AUDIO_EQUALIZER_MAX_NBANDS];

#include "../Filters/myFilters6x2_Stereo_Q15.h"

void FilterInputPlusEnergy(libq_q15_t XinLeft, libq_q15_t XinRight, libq_q15_t *YoutLeft,
libq_q15_t *YoutRight)
{
    libq_q15_t Yout0,Yout1,Yout2,Yout3,Yout4,Yout5,Yout6,Yout7;

    Yout0 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[0], XinLeft );
    AUDIO_EQUALIZER_BandEnergySumQ15[0] += abs(Yout0);

    Yout1 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[2], XinLeft );
    AUDIO_EQUALIZER_BandEnergySumQ15[1] += abs(Yout1);

    Yout2 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[4], XinLeft );
    AUDIO_EQUALIZER_BandEnergySumQ15[2] += abs(Yout2);

    Yout3 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[6], XinLeft );
    AUDIO_EQUALIZER_BandEnergySumQ15[3] += abs(Yout3);

    Yout4 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[8], XinLeft );
    AUDIO_EQUALIZER_BandEnergySumQ15[4] += abs(Yout4);

    Yout5 = AUDIO_EQUALIZER_Cascade2inQ15(&myFiltersLeft[10], XinLeft );
    AUDIO_EQUALIZER_BandEnergySumQ15[5] += abs(Yout5);
    /* Don't need these bands for 6 band filter
    Yout6 = AUDIO_EQUALIZER_Cascade2inQ15(&myFiltersLeft[12], XinLeft );
    AUDIO_EQUALIZER_BandEnergySumQ15[6] += abs(Yout6);

    Yout7 = AUDIO_EQUALIZER_Cascade2inQ15(&myFiltersLeft[14], XinLeft );
    AUDIO_EQUALIZER_BandEnergySumQ15[7] += abs(Yout7);
    */
    *YoutLeft = Yout0 + Yout1 + Yout2 + Yout3 + Yout4 + Yout5; // + Yout6 + Yout7;

    Yout0 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[0], XinRight );
    AUDIO_EQUALIZER_BandEnergySumQ15[8] += abs(Yout0);

    Yout1 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[2], XinRight );
    AUDIO_EQUALIZER_BandEnergySumQ15[9] += abs(Yout1);

    Yout2 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[4], XinRight );
    AUDIO_EQUALIZER_BandEnergySumQ15[10] += abs(Yout2);

    Yout3 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[6], XinRight );
    AUDIO_EQUALIZER_BandEnergySumQ15[11] += abs(Yout3);

    Yout4 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[8], XinRight );
    AUDIO_EQUALIZER_BandEnergySumQ15[12] += abs(Yout4);

    Yout5 = AUDIO_EQUALIZER_Cascade2inQ15(&myFiltersRight[10], XinRight );
    AUDIO_EQUALIZER_BandEnergySumQ15[13] += abs(Yout5);
    /* Don't need these bands for 6 band filter
    Yout6 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[12], XinRight );
    AUDIO_EQUALIZER_BandEnergySumQ15[14] += abs(Yout6);

    Yout7 = AUDIO_EQUALIZER_Cascade2inQ15(&myFiltersRight[14], XinRight );
    AUDIO_EQUALIZER_BandEnergySumQ15[15] += abs(Yout7);
    */
    *YoutRight = Yout0 + Yout1 + Yout2 + Yout3 + Yout4 + Yout5; // + Yout6 + Yout7;

    AUDIO_EQUALIZER_nSamples += 1;
}

```

Adjusting Band Filter Gains

```

#include "audio_equalizer.h"
#include "audio_equalizer_fixedpoint.h"

#include "myStereoFilters6x2_Q15.h"
#define NBANDS 6
#define NFILTERS 2

EQUALIZER_FILTER *pMyFilters;
GFX_EQUAL_CHANNEL myChannel;
uint8_t myBand;
int16_t iGainAdj;
libq_q0d15_t displayGainAdjQ15;

while ( !GainAdjDone )
{
    switch ( gainAdjStateGet() );
    {
        case GET_CHANNEL:
            myChannel = GetUserChannel();
            if ( GFX_EQUAL_CHANNEL_LEFT == myChannel )
            {
                pMyFilters = myFiltersLeft;
            }
            else
            {
                pMyFilters = myFiltersRight;
            }
            break;

        case GET_BAND:
            myBand = GetUserBand();
            GRAPHIC_EQUALIZER_ChannelFocus(true,myChannel,myBand);
            bandGain = // Base all gain adjustments on this old value
                AUDIO_EQUALIZER_FilterGainGetQ15(pMyFilters, NBANDS, NFILTERS,
                                                myBand, NFILTERS );

            break;

        case APPLY_GAIN_ADJ:
            iGainAdj = GetUserGainAdj(); // -50 dB <= iGainAdj <= +50 dB
            displayGainAdjQ15 = 327*iGainAdj + 16384;
            GRAPHIC_EQUALIZER_BandValueUpdate(GFX_EQUAL_FILTER_GAIN,channel,myBand,displayG
ainAdjQ15);

            adjBandGain = AUDIO_EQUALIZER_FilterGainAdjustQ15(bandGain,iGainAdj);
            AUDIO_EQUALIZER_FilterGainSetQ15(pMyFilters, NBANDS, NFILTERS,
                                            myBand, NFILTERS,
                                            adjBandGain );

            GRAPHIC_EQUALIZER_BandDisplayRefresh(myChannel,myBand);

            break;

        case GAIN_ADJUST_DONE:
            GRAPHIC_EQUALIZER_ChannelFocus(false,myChannel,myBand);
            GainAdjDone = true;
            break;

    } //end switch ( gainAdjStateGet() )

} //end while ( !GainAdjDone )

```

Customized Exception Handler

The source code in `audio_eq_exception-handler.c` can be added to the application's `main.c` file and customized to provide application-specific exception handling. In the example below, five LEDs are used to alert the user that an exception has occurred.

```
// *****
// *****
// Section: Exception handling
// *****
// *****
/*
```

The standard exception handling provided by the compiler's installation is insufficient to handle overflow exception that can happen inside of assembly IIR routines.

Exceptions are handled by the default code by simply throwing the application into a while (1) loop, which simply ends all processing of the application. This new code attempts to return control back to the application.

If an overflow exception is trapped a fallback return value is written to the `$v0` register for use by the IIR primitive that generated the overflow. The application can use +1 (0x7FFF) or Zero, or random noise as the fallback return value. See below.

```
*/
typedef struct _XCPT_FRAME
{
    uint32_t at;
    uint32_t v0;
    uint32_t v1;
    uint32_t a0;
    uint32_t a1;
    uint32_t a2;
    uint32_t a3;
    uint32_t t0;
    uint32_t t1;
    uint32_t t2;
    uint32_t t3;
    uint32_t t4;
    uint32_t t5;
    uint32_t t6;
    uint32_t t7;
    uint32_t t8;
    uint32_t t9;
    uint32_t ra;
    uint32_t lo;
    uint32_t hi;
    uint32_t cause;
    uint32_t status;
    uint32_t epc;
} XCPT_FRAME;

static enum {
    EXCEP_IRQ      = 0, // interrupt
    EXCEP_AdEL     = 4, // address error exception (load or ifetch)
    EXCEP_AdES     = 5, // address error exception (store)
    EXCEP_IBE      = 6, // bus error (ifetch)
    EXCEP_DBE      = 7, // bus error (load/store)
    EXCEP_Sys      = 8, // syscall
    EXCEP_Bp       = 9, // breakpoint
    EXCEP_RI       = 10, // reserved instruction
    EXCEP_CpU      = 11, // coprocessor unusable
    EXCEP_Overflow = 12, // arithmetic overflow
```

```

    EXCEP_Trap      = 13, // trap (possible divide by zero)
    EXCEP_IS1       = 16, // implementation specific 1
    EXCEP_CEU       = 17, // CorExtend Unuseable
    EXCEP_C2E       = 18, // coprocessor 2
} _excep_code;

/* EXCEPTION CODE TO LED Map *****
   LED #1 alerts that there has been an exception.
   The remaining LEDs show what type of exception has occurred.

Exception:      Index:  LEDs-
-----
EXCEP_IRQ       1      1 0001
EXCEP_AdEL      2      1 0010
EXCEP_AdES      3      1 0011
EXCEP_IBE       4      1 0100
EXCEP_DBE       5      1 0101
EXCEP_Sys       6      1 0110
EXCEP_Bp        7      1 0111
EXCEP_RI        8      1 1000
EXCEP_CpU       9      1 1001
EXCEP_Overflow  10     1 1010
EXCEP_Trap      11     1 1011
EXCEP_IS1       12     1 1100
EXCEP_CEU       13     1 1101
EXCEP_C2E       14     1 1110
Undefined       15     1 1111

***** */

static unsigned int _excep_code;
static unsigned int _excep_addr;

#if defined(USE_NEW_EXCEPTION_HANDLER)
void __attribute__((nomips16)) _general_exception_handler (XCPT_FRAME* const pXFrame)
{
    _excep_addr = pXFrame->epc;
    _excep_code = pXFrame->cause;
    _excep_code = (_excep_code & 0x0000007C) >> 2;

    // Report Exception using LEDs, port is grounded to light an LED
    PORTACLR = 1<<4; // Turn on "exception alert" LED
    switch ( _excep_code ) // Identify the exception
    {
        case EXCEP_IRQ: // RA5    RA6    RA7    RA9
            PORTACLR = (0<<5)+(0<<6)+(0<<7)+(1<<9);
            break;

        case EXCEP_AdEL:
            PORTACLR = (0<<5)+(0<<6)+(1<<7)+(0<<9);
            break;

        case EXCEP_AdES:
            PORTACLR = (0<<5)+(0<<6)+(1<<7)+(1<<9);
            break;

        case EXCEP_IBE:
            PORTACLR = (0<<5)+(1<<6)+(0<<7)+(0<<9);
            break;

        case EXCEP_DBE:
            PORTACLR = (0<<5)+(1<<6)+(0<<7)+(1<<9);
            break;
    }
}

```

```

    case EXCEP_Sys:
        PORTACLR = (0<<5)+(1<<6)+(1<<7)+(0<<9);
        break;

    case EXCEP_Bp:
        PORTACLR = (0<<5)+(1<<6)+(1<<7)+(1<<9);
        break;

    case EXCEP_RI:
        PORTACLR = (1<<5)+(0<<6)+(0<<7)+(0<<9);
        break;

    case EXCEP_CpU:
        PORTACLR = (1<<5)+(0<<6)+(0<<7)+(1<<9);
        break;

    case EXCEP_Overflow:
        PORTACLR = (1<<5)+(0<<6)+(1<<7)+(0<<9);
        break;

    case EXCEP_Trap:
        PORTACLR = (1<<5)+(0<<6)+(1<<7)+(1<<9);
        break;

    case EXCEP_IS1:
        PORTACLR = (1<<5)+(1<<6)+(0<<7)+(0<<9);
        break;

    case EXCEP_CEU:
        PORTACLR = (1<<5)+(1<<6)+(0<<7)+(1<<9);
        break;

    case EXCEP_C2E:
        PORTACLR = (1<<5)+(1<<6)+(1<<7)+(0<<9);
        break;

    default:
        PORTACLR = (1<<5)+(1<<6)+(1<<7)+(1<<9);
        break;
} //end switch ( _excep_code )

// Report exception via UART.
sprintf(ioString, " EXCEPTION: %d at %08x :EXCEPTION \r\n", _excep_code, _excep_addr);
SendDataBuffer(ioString, strlen(ioString) );

if (_excep_code == EXCEP_Overflow)
{
    // Provide fallback return value for filtering primitive throwing an overflow exception.
    pXFrame->v0 = 0x7FFF; // set function output to maximum (saturation)
    pXFrame->v1 = 0x7FFF; // set intermediate results to maximum (saturation)
    pXFrame->epc = pXFrame->epc + 4; // set return from exception to next instructino
(skip)
}

return;

// Double CRAP! The exception handler has thrown an exception!!
sprintf(ioString, " EXCEPTION:EXCEPTION: %d at %08x :EXCEPTION:EXCEPTION \r\n", _excep_code,
_excep_addr);
SendDataBuffer(ioString, strlen(ioString) );

while (1) {
    // Wait for the cavalry to arrive...
    asm("NOP");
}
}

```

1.8 Equalization Filters

1.8.1 Example Filter Definition Files

Example Filters

In the folder `./framework/math/audio_equalizer/filters` you will find predefined filters. Filters are defined in `.h` files that can be `#included` in application source code. (See the code examples in [AUDIO_EQUALIZER_BandEnergyUpdateQ15](#) and [AUDIO_EQUALIZER_BandEnergyNSamplesSet](#).)

GraphicEqualizer...

Files starting with `GraphicEqualizer...` were designed in Matlab using the `GraphicEqualizerDesign.m`. File names are of the form

```
GraphicEqualizer{NfreqBands}x{NfiltersPerBand}_Q{15|31}{.h|.jpg|.mat}.
```

The `dot h` file initializes a filter structure so that the filtering [library](#) can use the filters defined in the file. The `dot MAT` file contains the filter workspace used to design the filters and can be used in validating the filter on PIC32 devices. The `dot JPG` files shows the designed filter response, as calculated by the script that designed the filter coefficients.

myFilters...

Files starting with `myFilters...` were designed in Matlab using the `GraphicEqualizerFilterDesignScript.m`. File names are of the form

```
myFilters{NfreqBands}x{NfiltersPerBand}_Q{15|31}{.h|.jpg|.mat}.
```

The `dot h` file initializes a filter structure so that the filtering [library](#) can use the filters defined in the file. The `dot MAT` file contains the filter workspace used to design the filters and can be used in validating the filter on PIC32 devices. The `dot JPG` file shows the designed filter response, as calculated by the script that designed the filter coefficients.

ParametricFilters...

Files starting with `ParametricFilter...` were designed in Matlab using the `ParametricEqualizerDesign.m`. File names are of the form

```
ParametricFilter1x{Nfilters}_Q{15|31}{.h|.jpg|.mat}.
```

The `dot h` file initializes a filter structure so that the filtering [library](#) can use the filters defined in the file. The `dot MAT` file contains the filter workspace used to design the filters and can be used in validating the filter on PIC32 devices. The `dot JPG` file shows the designed filter response, as calculated by the script that designed the filter coefficients.

1.8.2 Matlab/Octave

Filter Design Tools

Design of new filters is supported by Matlab(tm) and Matlab(tm)/Octave scripts that are provided along with the filtering [library](#). These tools can design filters for both graphic equalizers and parametric equalizers.

Matlab

Matlab is available from Mathworks (<http://www.mathworks.com>). An additional toolbox (*Signal Processing Toolbox*) is needed to run the Matlab-only graphic equalizer filter script `GraphicEqualizerFilterDesignScript.m`. Additionally, the filter design scripts `GraphicEqualizerDesign.m` and `ParametricEqualizerDesign.m` will run from Matlab(tm).

GNU Octave

GNU Octave is a freeware clone of Matlab and supports basically all Matlab primitives. It has many add-on packages for subjects such as DSP and Mechanical Engineering. Octave only supports a command line (> prompt) interface, but there are GUIs that will run on top of Octave to provide a similar look and feel to Octave as that of Matlab. The filter design scripts `GraphicEqualizerDesign.m` and `ParametricEqualizerDesign.m` will run from GNU Octave as well as Matlab(tm).

GNU Octave GUIs

Octave's user interface is a command window, not a GUI. Several GUIs are available to run on top of Octave so that the user experience is closer to Matlab's.

- GUI Octave is a freeware GUI, written by Joaquim Varandas. It is available at <http://gui octave.software.informer.com>.
- Xoctave is a commercial product, available at <http://www.xoctave.com>.

How To Install GNU Octave:

(Original instructions found here: http://wiki.octave.org/Octave_for_Windows)

1. Download the Octave Windows and Octave Packages binaries at:

<http://sourceforge.net/projects/octave/files/Octave%20Windows%20binaries/Octave%203.6.4%20for%20Windows%20MinGW%20installer/>

2. The two files to download are: `Octave3.6.4_gcc4.6.2_20130408.7z` `Octave3.6.4_gcc4.6.2_pkgs_20130402.7z`

3. Create the directory `C:\Octave`

4. Copy both downloaded archives to `C:\Octave`

5. Right click on each *.7z file in `C:\Octave` and select 7-Zip>>Extract Here

6. Copy the following lines into the Octave window and execute them:

```
pkg rebuild -auto
pkg rebuild -noauto ad
pkg rebuild -noauto nan
pkg rebuild -noauto gsl
pkg rebuild -auto java
```

7. Enlarge the font in Octave by clicking on the icon in the upper left corner of the window and select Properties. Select font size (18 pt)

1.8.3 Graphic Equalization Filter Design Tools

Introduction

The Harmony folder `apps\filters\audio\filter_design` contains filter design tools in Matlab(tm)/Octave that support designing new equalization filters.

Designing Filters for Graphic Equalizers Using Matlab(tm)

The Matlab(tm) script **GraphicEqualizerFilterDesignScript.m** uses the `yulewalk` function (*part of the Signal Processing Toolbox*) to design equally spaced frequency bands between DC and the folding frequency ($F_0 = F_s/2$). Band edges are specified in the vector `fBands`, with frequencies normalized by F_0 . The desired filter amplitude is specified by `mBandsN`, where N specifies the frequency band:

```
% Filter specification setups *****
switch ( nBands )
.
.
.
case {5}
    % Frequencies, including band center and band edges
    %           Edge  Cntr  Edge  Cntr  Edge  Cntr  Edge  Cntr  Edge  Cntr  Edge
    fBands = [    0    .1    .2    .3    .4    .5    .6    .7    .8    .9    1 ];

    % Band amplitude desired, with 6 dB at band edges
    mBand1 = [    1    1    .5    0    0    0    0    0    0    0    0 ];
    mBand2 = [    0    0    .5    1    .5    0    0    0    0    0    0 ];
    mBand3 = [    0    0    0    0    .5    1    .5    0    0    0    0 ];
    mBand4 = [    0    0    0    0    0    0    .5    1    .5    0    0 ];
    mBand5 = [    0    0    0    0    0    0    0    0    .5    1    1 ];

    % Peak amplitude for each band
    peakAmp = [    +1.5        +1            +1            +1            +1.5        ];

    % Sign used for each band output, alternating signs prevents band edges from
    % being 180 out of phase and producing notch in overall filter response.
    sBands = [    +1            -1            +1            -1            +1        ];

    mBands = [ mBand1; mBand2; mBand3; mBand4; mBand5 ];
```

The vector `peakAmp` provides adjustments for the first and last filter bands, so that the overall filter response is as flat as possible.

Band edges can be adjusted if equally-spaced bands are not desired.

Designing Filters for Graphic Equalizers Using Matlab/Octave

The Harmony folder `apps\filters\audio\filter_design` contains filter design tools in Matlab(tm)/Octave that support designing parametric equalization filters. Launch Matlab(tm)/Octave, change the default directory to the location of the filter design scripts, and then start the script by entering **GraphicEqualizerDesign** followed by a return:

```
Welcome to Xoctave 3.3.
Please visit http://www.xoctave.com to get informed about updates and announcements.
>>
>> cd( 'C:\Harmony\apps\filters\audio\filter_design' )
>> GraphicEqualizerDesign % for Graphic Equalizer filters
>>
```

A dialog window will appear:

Graphic Equalization Filter Design 1

Filter_Bit_Width: 16

Sample_Rate_Hz: 44100

#_Filters/Band: 2

Center_Frequencies_Hz:					
0	4410	8820	13230	17640	21000

Gains_dB:					
1	0	0	0	-1	0.5

Bandwidths_Hz:					
6000	4750	5500	5500	4500	4500

Signs [+1/-1]:					
-1	-1	1	-1	1	-1

Qs [LowPass,HighPass]:					
0.717	NA	NA	NA	NA	0.717

Plot_Command [plot,semilogx]: plot

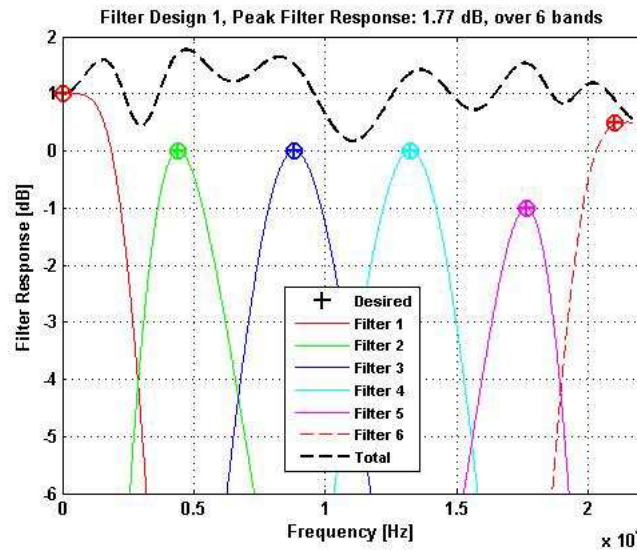
OK Cancel

(Note screen shots are from Matlab(tm). Octave produces a slightly different dialog window that is functionally identical.)

The dialog window has inputs for:

- Filter Bit Width (either 16 or 32)
- Sample Rate in Hz
- Number of Biquad IIRs in cascade for each frequency band
- Center Frequencies for Each Filter, in Hz
- Desired Gain for Each Filter, in dB
- Bandwidth, in Hz, for Each Filter
- Signs (+1/-1) used to add filter outputs together
- Shelving Qs for the first (lowpass) and last (highpass) filters
- Plot Command, which determines whether the frequency axis is linear (plot) or logarithmic (semilogx)

Frequencies, gains, bandwidths, signs, and Qs are lined up so that it is easy to edit a filter's parameters. To change the number of filters used simply delete a column of data. When the parameters are set, select "OK" to generate the filters and display a plot:

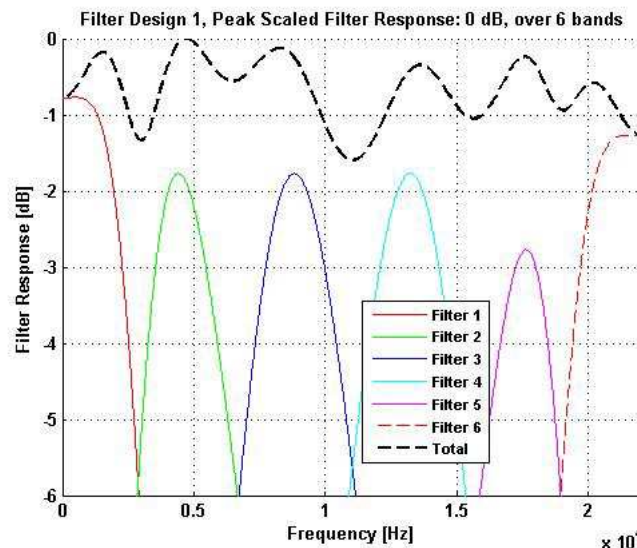


Next a dialog window appears asking if you are done:

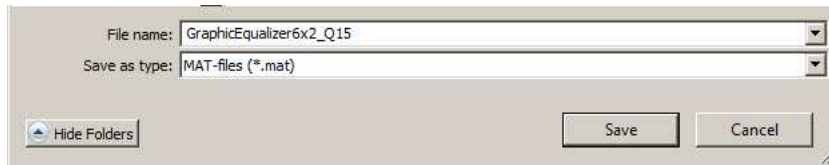


Select "No" to return to the parameters dialog screen for additional tweaking or "Yes" to move on to saving the filters just generated.

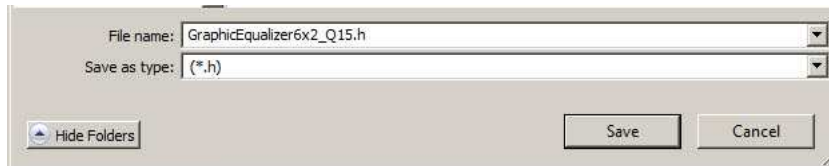
If you select "Yes" then a plot of the scaled filter response is shown, where the total filter response has been normalized to a peak gain of unity (0 dB). This is the filter setup that is saved for use in PIC32 firmware.



Next a dialog window appears to save the filter design workspace for later reuse in designing more filters or in validating the filters on PIC32 devices. The window allows you to save the .MAT file anywhere on your computer:



Next you are prompted to save the .h file, which defines the filters for PIC32 firmware:



To help you keep track of each iteration of filter design the parameters used at each iteration are dumped to the Matlab(tm)/Octave console:

```

FILTER DESIGN 1 *****
nFilterBits: 16, Fs: 44100 Hz, # Bands: 6
Center Freqs [Hz]:      0      4410      8820      13230      17640
21000
Filter Gains [dB]:      1      0      0      0
-1      0.5
Filter BWidth [Hz]:    6000      4750      5500      5500
4500      4500
LowPass/HighPass Qs:   0.717      NA      NA      NA      NA
0.717

```

You can load the setup used for previous filters by loading the workspace .mat file into Matlab(tm)/Octave before launching the design script.

1.8.4 Parametric Equalization Filter Design

Designing Filters for Parametric Equalizers

The Harmony folder `apps\filters\audio\filter_design` contains filter design tools in Matlab(tm)/Octave that support designing parametric equalization filters. Launch Matlab(tm)/Octave, change the default directory to the location of the filter design scripts, and then start the script by entering **ParametricEqualizerDesign** followed by a return:

```

Welcome to Xoctave 3.3.
Please visit http://www.xoctave.com to get informed about updates and announcements.
>>
>> cd( 'C:\Harmony\apps\filters\audio\filter_design' )

>> ParametricEqualizerDesign % for Parametric Equalizer filters
>>

```

A dialog window will appear:

Parametric Equalization Filter Design

Filter_Bit_Width: 32

Sample_Rate_[Hz]: 44100

Center_Frequencies_[Hz]:

125	250	500	1000	2000	4000	8000	12000
-----	-----	-----	------	------	------	------	-------

Gains_[dB]:

1	0.5	2	1	-1	-2	-2	-2
---	-----	---	---	----	----	----	----

Bandwidths_[Hz]:

NA	176.777	353.553	707.107	1414.21	2828.43	5656.85	NA
----	---------	---------	---------	---------	---------	---------	----

Shelving_Qs_[Low,High]:

0.707107	NA	NA	NA	NA	NA	NA	0.707107
----------	----	----	----	----	----	----	----------

Plot_Command_[plot,semilogx]: semilogx

OK Cancel

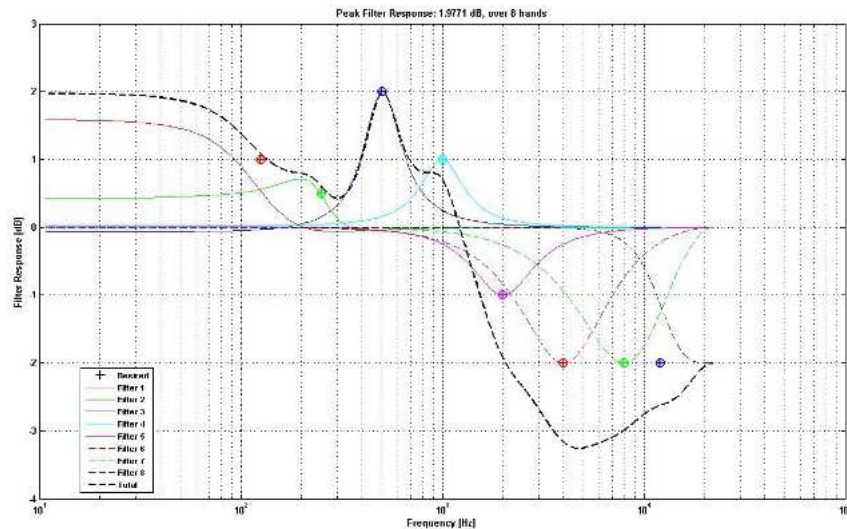
These defaults were first loaded into Matlab(tm)/Octave from a file before executing the script. The file was

C:\Harmony\framework\math\audio_equalizer\filters\ParametricFilters1x8_Q31.mat

which was produced when the script was used to generate the [ParametricFilters1x8_Q31.h](#) file in the same directory. The dialog window has inputs for:

- Filter Bit Width (either 16 or 32)
- Sample Rate in Hz
- Center Frequencies for Each Filter, in Hz
- Desired Gain for Each Filter, in dB
- Bandwidth, in Hz, for Each Interior Filter
- Shelving Qs for the first and last filter.
 - $Q = 1/\sqrt{2}$ provides maximally flat pass band up to the cutoff frequency.
 - $Q < 1/\sqrt{2}$ provides higher pass band attenuation
 - $Q > 1/\sqrt{2}$ provides additional gain around the cutoff frequency
- Plot Command, which determines whether the frequency axis is linear (plot) or logarithmic (semilogx)

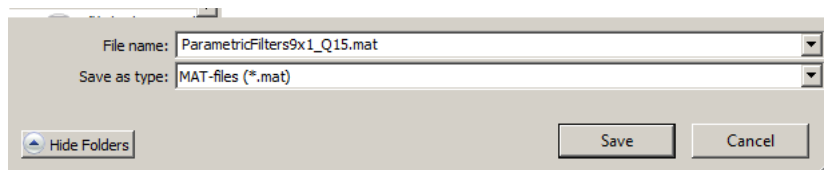
Frequencies, gains, bandwidths, and Qs are lined up so that it is easy to edit a filter's parameters. To change the number of filters used simply delete a column of data. When the parameters are set, select "OK" to generate the filters and display a plot:



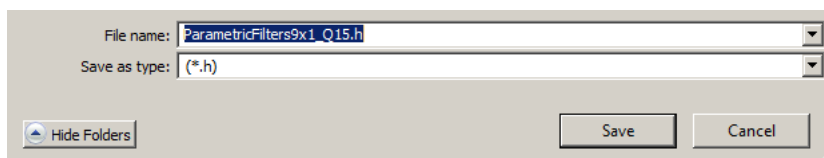
Next a dialog window appears asking if you are done:



Select "No" to return to the parameters dialog screen for additional tweaking or "Yes" to move on to saving the filters just generated. If you select "Yes" then a dialog window appears to save the filter design workspace for later reuse in designing more filters or in validating the filters on PIC32 devices. The window allows you to save the .MAT file anywhere on your computer:



Next you are prompted to save the .h file, which defines the filters for PIC32 firmware:



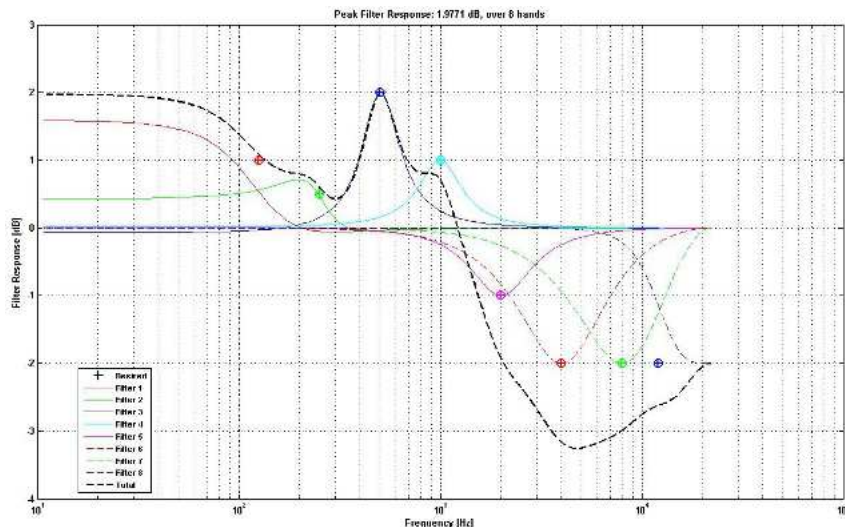
To help you keep track of each iteration of filter design the parameters used at each iteration are dumped to the Matlab(tm)/Octave console:

```

FILTER DESIGN *****
nFilterBits: 16, Fs: 44100 Hz, # Bands: 8
Center Freqs [Hz]:      125      250      500      1000
2000      4000      8000      12000
Filter Gains [dB]:      1      0.5      2      1
-1      -2      -2      -2
Filter BWidth [Hz]:      NA      176.777      353.553      707.107
1414.21      2828.43      5656.85      NA
Shelving Qs:      0.707107      NA      NA      NA
NA      NA      NA      0.707107
  
```


A Warning About Filter Bit Widths

If you simply change the filter bit width from 32 to 16 in the above setup, and generate 16-bit wide filters instead of 32-bit, you will find that the behavior of the first two filters changes dramatically simply because of rounding the coefficients to fixed point values with 16 instead of 32 bits:



The takeaway from this is that some types of parametric filters, especially those centered at low frequencies, are very sensitive to coefficient rounding. But since using 32 bit filters for an 8-filter setup is only 23% more computationally expensive than the same setup with 16-bit filters, it is usually not necessary to design and use 16-bit filters.

Note that the change in filter behavior is caused solely by the reduction in coefficient bit width, not because the filters are calculated using 16-bit math instead of 32-bit. The file

C:\Harmony\framework\math\audio_equalizer\filters\ParametricFilters1x8_Q31_Hacked.h

contains 16-bit coefficients scaled up to Q31 for Filters 1 and 2. These coefficients can be run from the 32-bit validation project and the same validation plot produced as shown above.

1.8.5 Filter Validation Tools

Introduction

Equalization filter designed with Matlab(tm)/Octave or some other tool must be validated on the target hardware using target signals to ensure that the filters perform as expected. The folder `./apps/filters/audio/filter_validation` contains Matlab(tm)/Octave scripts and PIC2MX firmware in support of this task.

Out of the box, MPLAB.x validation projects will run on PIC32 Bluetooth Audio Development Board. Projects with a `_Q15.X` suffix supports 16 bit (Q0.15 or Q15) filters while `_Q31.X` projects supports 32 bit (Q0.31 or Q31) filters:

- **ParametricFilterValidation_Q15.X** - Parametric filter validation for Q15 (16-bit) filters
- **ParametricFilterValidation_Q31.X** - Parametric filter validation for Q31 (32-bit) filters
- **GFXFilterValidation_Q15.X** - Graphic Equalizer filter validation for Q15 (16-bit) filters
- **GFXFilterValidation_Q31.X** - Graphic Equalizer filter validation for Q31 (32-bit) filters

On the target board UART4 is used to transmit text data back to a PC, which captures the text into an ASCII flat file using Hyperterminal, RealTerm, or some other terminal emulation utility. The captured text is saved to a text file, which is analyzed

using the Matlab(tm)/Octave script `ValidateFilterResponseScript.m` or `ValidateParametricFilterResponse.m`, located in the folder `./Matlab-Octave`.

A UART4 transmit pin can be found on pin 4 of the J4 connector on the PIC2 Bluetooth Audio Development Board. This pin can be connected to the receive pin of a PICKit Serial Analyzer (PKSA) that has been programmed for USART communication. A typical bench-top setup is shown below:



Collecting Filter Data

The filters to be analyzed are specified at the top of the file `validation_tool_Q15.c` (or `_Q31.c`) or `parametric_filter_validation_tool.c`:

```
#include "math/audio_equalizer/filters/myFilters6x2_Q15.h"
// #include "../Filters/myBadFilters6x2_Q15.h"
#define NUM_BANDS 6
#define NFILTERS_PER_BAND 2
```

(The filters in the commented out file `myBadfilters6x2.h` have incorrect gain settings and can be used to demonstrate that the exception handler works correctly.) The file `myFilters6x2_Q15.h` contains filter coefficients for a 6-band graphic equalizer, where each band has two IIR biquads for filtering using 16-bit coefficients.

The input signal used to validate the filters is specified by the sampling frequency and FFT size:

```
// Variables for measuring filter response: Sampling Frequency and FFT size
double Fs = 44100; // sampling frequency, in Hz
uint16_t nFFT = 1024; // FFT size
```

A linear FM chirp is used as input to the filter bank with frequencies sweeping from DC (1st FFT bin) up to the folding frequency ($F_s/2$). At each frequency $nFFT$ samples are computed and filtered. The amplitude of this signal is specified by:

```
// Parameters for input tone generation
// double ampXin = 0.891240938; // Input waveform amplitude -1 dBFS
double ampXin = 1.0;
```

Gain adjustments for each of the Graphic Equalizer frequency bands can be specified by:

```
// Gain adjustments
EQUALIZER_FILTER_GAIN bandGain,adjBandGain;
int16_t myGainAdjustments[] = { 0, 0, 0, 0, 0, 0, 0, 0 };
//int16_t myGainAdjustments[] = { -1, -1, -1, -1, -1, -1, -1, -1 };
//int16_t myGainAdjustments[] = { 0, 0, 0, 0, -2, 0, 0, 0 };
```

with the gains specified in integer dB's. (*Parametric Equalizer gains are not adjustable, since to adjust a parametric equalizer gain would change the gain out of band from unity to some other value.*) The output of each band's filters and the overall filter output is calculated and the filter response is calculated using a Discrete Fourier Transform (DFT) for each band and the overall filter. This data is pumped out the UART transmit pin for each FFT bin:

```
// Dump filter response results out UART to Matlab or Octave
for (iBand = 0; iBand < nBands+1; iBand++)
{
    BandEnergy = AUDIO_EQUALIZER_BandEnergyGetQ15(iBand,true)/65536.0;
    sprintf(ioString,"%d,%g,%g,%g,%g\r\n",
            iBand,Fc,YoutAmpSqr[iBand],YoutPhase[iBand],(iFreq==0 ? 1.0 :
2.0)*BandEnergy);
    // BandEnergy = A^2/2, where A = signal amplitude
    SendDataBuffer(ioString, strlen(ioString) );
}
```

The DFT result is output as an amplitude squared and as phase (in degrees) for use by Matlab(tm)/Octave in comparing the measured results with filter responses calculated when the filters were designed. This comparison is accomplished using the script `ValidateFilterResponseScript.m`.

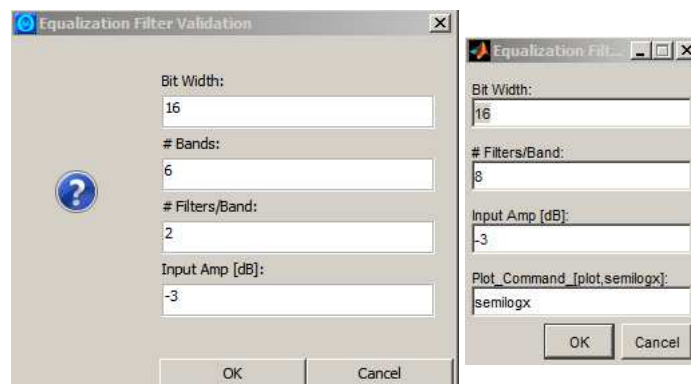
Validating the Data

Launch Matlab(tm) or Octave. Change the default directory to the location of the `ValidateFilterResponseScript` (Graphic Equalizer filters) and `ValidateParametricFilterResponse` (Parametric Equalizer filters) and execute the script corresponding to your filters:

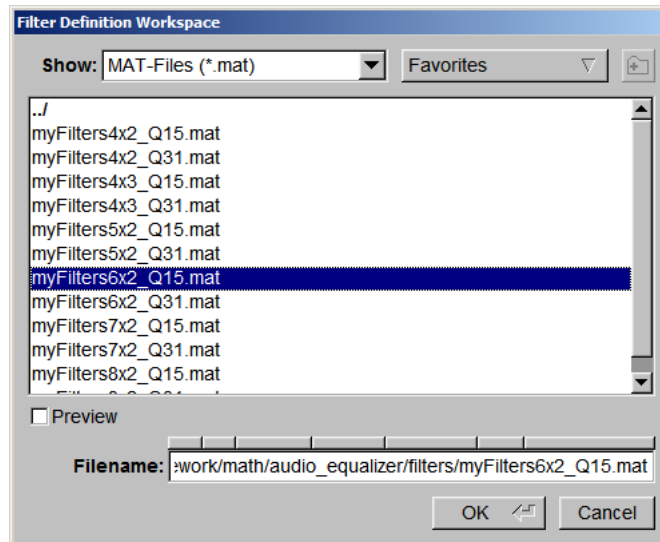
```
Welcome to Xoctave 3.3.
Please visit http://www.xoctave.com to get informed about updates and announcements.
>>
>> cd( 'C:\Harmony\apps\filters\audio\filter_design' )

>> ValidateFilterResponseScript % for Graphic Equalizer filters
>> % Or ValidateParametricFilterResponse for Parametric Equalizer filters
```

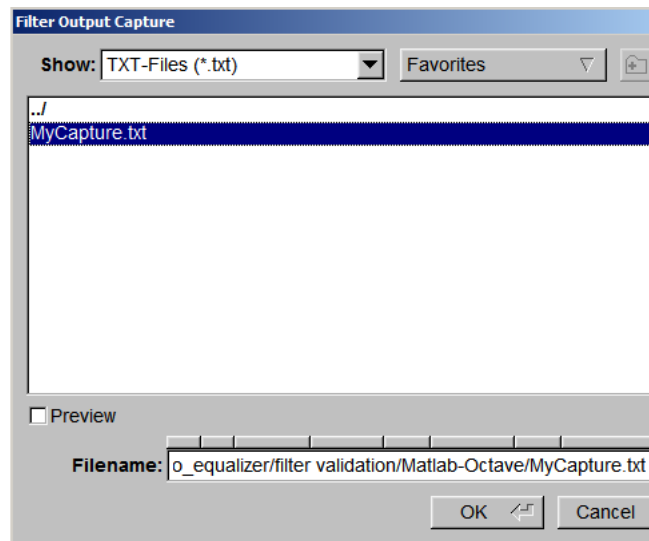
A dialog window will appear. Input the filter bit width (16 or 32), the number of frequency bands in the filter bank, the number of IIR biquad filters per band, and the input amplitude, in dB. For Parametric Equalization filters there is an additional input, which determines whether the frequency axis is linear (plot) or logarithmic (semilogx).



After editing the default values, press "OK" to continue. Next load the captured file, first loading the .MAT file belonging to the filter:



Then load the captured text file:



The script will then plot actual versus desired filter responses for each filter band and the overall filter response.

As described in the validation script, the captured data has the following format:

```
%Format of CapturedDataArray, for a 6 band filter setup
%
% Band| Freq   | Filt Amp^2 | Filt Phase | Yout Mean Squared
% 0, 43.0664, 0.877196 , 1.56604, 0.87616
% 1, 43.0664, 1.43779e-05, 104.977, 0
% 2, 43.0664, 7.5499e-08, -90.0112, 0
% 3, 43.0664, 0.000512932, -179.669, 0.000427246
% 4, 43.0664, 1.12161e-08, -108.557, 0
% 5, 43.0664, 8.56599e-06, -179.652, 0
% 6, 43.0664, 0.828278 , 1.81478, 0.827271
%
% Band outputs are indexed from 0 to nBands, index = nBands is total response
%
```

Sometimes a null (\0) character is captured at the start of the text file. This produces an error in Matlab(tm)/Octave when the file is read:

```
Error using load
Unknown text on line number 1 of ASCII file
C:\Harmony\apps\examples\math\audio_equalizer\filter
validation\Matlab-Octave\capture.txt
" ".
Error in ValidateParametricFilterResponse (line 88)
load(PathCapturedFile);
```

It occurs because a null character has been captured in the first line of the file:

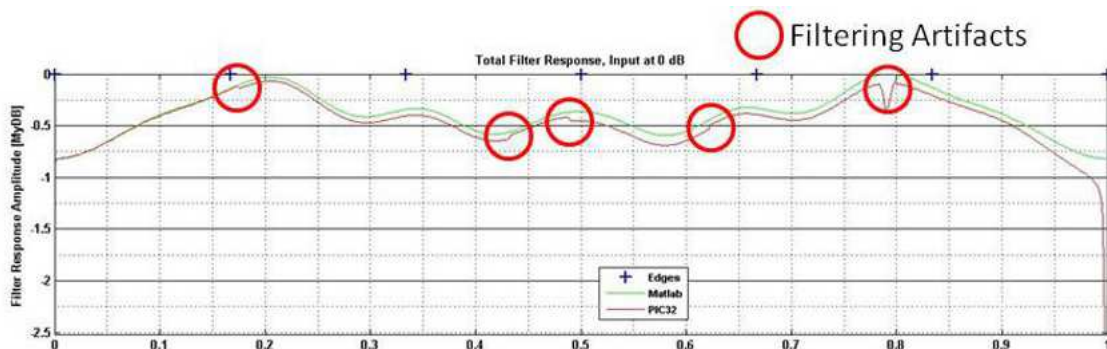
```
0,0,0.707332,0,0
0,43.0664,0.708033,-3.51642,0
0,86.1328,0.650234,-6.80699,0
0,129.199,0.56688,-7.64474,0
```

With any available text editor simply delete the first character in the file. This will allow Matlab(tm)/Octave to read and process the edited file.

It is best to capture data (using Hyperterminal or RealTerm) to a local instead of network file. Network latency accessing a remotely located file can cause dropped characters in the data capture, which causes Matlab(tm)/Octave to error out when trying to read the captured data text file.

A Warning About Truncation and Overflows

The validation testbench software provided will light up the LEDs on the Bluetooth Audio Development board whenever the filtering software throws an exception. (See [Core Exception Handling](#).) But there are cases where filtering artifacts will occur without exceptions. A case in point occurs when filters normalized with peak of 0 dB (unity) gain are drive with signals at 0 dBFS. If the signal is at or near the frequency of the peak filter response "**interesting**" things can happen without the software throwing an overflow exception. Here is an example:



Note there are filtering artifacts not only at the peak of the filter's gain, but at other frequencies as well. The takeaway: **Never override the filters!**

1.8.6 A Warning About Stereo Filters

WARNING*WARNING*WARNING:

While it is clearly recommended to use the same filter design for both left and right stereo channels, **you cannot use the same EQUALIZER FILTER (or EQUALIZER FILTER 32) structure for both channels**. This is because filter memory is part of the structure and you cannot share filter memory between channels. Thus there must be a dedicated filter structure for both left and right channels.

(Note also that having a filter structure for the left channel and one for the right channel allows different band filter gains between the channels.)

Correct:

```

#include <stdint.h>
#include <stdbool.h>
#include <math.h>

#include "math/audio_equalizer/audio_equalizer.h"

#include "myFilters8x2Stereo_Q15.h"

void FilterInput(libq_q15_t XinLeft, libq_q15_t XinRight, libq_q15_t *YoutLeft, libq_q15_t
*YoutRight)
{
    libq_q15_t YoutLeft0,YoutLeft1,YoutLeft2,YoutLeft3,YoutLeft4,YoutLeft5,YoutLeft6,YoutLeft7;
    libq_q15_t
YoutRight0,YoutRight1,YoutRight2,YoutRight3,YoutRight4,YoutRight5,YoutRight6,YoutRight7;

    YoutLeft0 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[0], XinLeft );
    YoutRight0 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[0], XinRight );

    YoutLeft1 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[2], XinLeft );
    YoutRight1 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[2], XinRight );
    .
    .
    .
    YoutLeft5 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[10], XinLeft );
    YoutRight5 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[10], XinRight );

    *YoutLeft = YoutLeft0 + YoutLeft1 + YoutLeft2 + YoutLeft3 + YoutLeft4 + YoutLeft5;
    *YoutRight = YoutRight0 + YoutRight1 + YoutRight2 + YoutRight3 + YoutRight4 + YoutRight5;

}

```

Incorrect:

```

#include <stdint.h>
#include <stdbool.h>
#include <math.h>

#include "math/audio_equalizer/audio_equalizer.h"

#include "myFilters8x2_Q15.h"

void FilterInput(libq_q15_t XinLeft, libq_q15_t XinRight, libq_q15_t *YoutLeft, libq_q15_t
*YoutRight)
{
    libq_q15_t YoutLeft0,YoutLeft1,YoutLeft2,YoutLeft3,YoutLeft4,YoutLeft5,YoutLeft6,YoutLeft7;
    libq_q15_t
YoutRight0,YoutRight1,YoutRight2,YoutRight3,YoutRight4,YoutRight5,YoutRight6,YoutRight7;

    YoutLeft0 = AUDIO_EQUALIZER_Cascade2inQ15( &myFilters[0], XinLeft );
    YoutRight0 = AUDIO_EQUALIZER_Cascade2inQ15( &myFilters[0], XinRight );

    YoutLeft1 = AUDIO_EQUALIZER_Cascade2inQ15( &myFilters[2], XinLeft );
    YoutRight1 = AUDIO_EQUALIZER_Cascade2inQ15( &myFilters[2], XinRight );
    .
    .
    .
    YoutLeft5 = AUDIO_EQUALIZER_Cascade2inQ15( &myFilters[10], XinLeft );
    YoutRight5 = AUDIO_EQUALIZER_Cascade2inQ15( &myFilters[10], XinRight );

    *YoutLeft = YoutLeft0 + YoutLeft1 + YoutLeft2 + YoutLeft3 + YoutLeft4 + YoutLeft5;

```

```
*YoutRight = YoutRight0 + YoutRight1 + YoutRight2 + YoutRight3 + YoutRight4 + YoutRight5;  
}
```

1.9 Library Interface

1) Filter Routines In C

	Name	Description
≡	AUDIO_EQUALIZER_IIRinQ15andC	Applies equalization filter defined by *pFilter to Xin and provides single output.
≡	AUDIO_EQUALIZER_IIRinQ15FastC	Applies equalization filter defined by *pFilter to Xin and provides single output.
≡	AUDIO_EQUALIZER_IIRinQ31andC	Applies equalization filter defined by *pFilter to Xin and provides single output.

2) Filter Routines In Assembly

	Name	Description
≡	AUDIO_EQUALIZER_IIRinQ15	Applies equalization filter defined by *pFilter to Xin and provides single output.
≡	AUDIO_EQUALIZER_IIRinQ31	Applies equalization filter defined by *pFilter to Xin and provides single output.

3) Single Band, Cascade of IIRs, in Assembly







	Name	Description
≡	AUDIO_EQUALIZER_Cascade2inQ15	Performs a single output of a cascade of 2 biquad IIR filters.
≡	AUDIO_EQUALIZER_Cascade2inQ31	Performs a single output of a cascade of 2 biquad IIR filters.
≡	AUDIO_EQUALIZER_Cascade8inQ15	Performs a single output of a cascade of 8 biquad IIR filters.
≡	AUDIO_EQUALIZER_Cascade8inQ31	Performs a single output of a cascade of 8 biquad IIR filters.

4) Multiple Bands, Multiple Filters per Band, in Assembly







	Name	Description
≡	AUDIO_EQUALIZER_Parallel4x2inQ15	Performs 4 parallel IIR filters, with 2 series biquad filters each, and sums the result.
≡	AUDIO_EQUALIZER_Parallel4x2inQ31	Performs 4 parallel IIR filters, 2 series biquad filters each, and sums the result.
≡	AUDIO_EQUALIZER_Parallel8x2inQ15	Performs 8 parallel IIR filters, with 2 series biquad filters each, and sums the result.
≡	AUDIO_EQUALIZER_Parallel8x2inQ31	Performs 8 parallel IIR filters, with 2 series biquad filters each, and sums the result.
≡	AUDIO_EQUALIZER_ParallelNx2inQ15	Performs N parallel IIR filters, 2 series biquad filters each, and sums the result.
≡	AUDIO_EQUALIZER_ParallelNx2inQ31	Performs N parallel IIR filters, 2 series biquad filters each, and sums the result.
≡	AUDIO_EQUALIZER_ParallelNxMinQ15	Performs N parallel IIR filters, M series biquad filters each, and sums the result.
≡	AUDIO_EQUALIZER_ParallelNxMinQ31	Performs N parallel IIR filters, M series biquad filters each, and sums the result.

5) Filter Gain Routines










	Name	Description
≡	AUDIO_EQUALIZER_FilterGainAdjustQ15	Adjusts a filter gain structure by the integer gain adjustment provided
≡	AUDIO_EQUALIZER_FilterGainAdjustQ31	Adjusts a filter gain structure by the integer gain adjustment provided

	AUDIO_EQUALIZER_FilterGainGetQ15	Gets the filter gain for a given band and filter.
	AUDIO_EQUALIZER_FilterGainSetQ15	Gets the filter gain for a given band and filter.
	AUDIO_EQUALIZER_FilterGainGetQ31	Gets the filter gain for a given band and filter.
	AUDIO_EQUALIZER_FilterGainSetQ31	Gets the filter gain for a given band and filter.
	AUDIO_EQUALIZER_GainNormalizeQ15	Normalize all the EQUALIZER_FILTER_GAIN 's in a filter array so that the gains can be applied correctly by each filtering function.
	AUDIO_EQUALIZER_GainNormalizeQ31	Normalize all the EQUALIZER_FILTER_GAIN 's in a filter array so that the gains can be applied correctly by each filtering function.








6) Band Energy Estimation

	Name	Description
	AUDIO_EQUALIZER_BandEnergySumsInit	Initialize band energy measurements, clearing band energy sum array and number of energy samples for each band.
	AUDIO_EQUALIZER_BandEnergyNSamplesSet	Resets number of samples used to update band energy measurements.
	AUDIO_EQUALIZER_BandEnergyUpdateQ15	Update band energy estimate for a given filter band with new filter output. "Q15" suffix designates this routine is for signals with Q15 fixed point format.
	AUDIO_EQUALIZER_BandEnergyUpdateQ31	Update band energy estimate for a given filter band with new filter output. "Q31" suffix designates this routine is for signals with Q31 fixed point format.
	AUDIO_EQUALIZER_BandEnergyGetQ15	Get band energy estimate for a given filter band. "Q15" suffix designates this routine is for signals with Q15 fixed point format.
	AUDIO_EQUALIZER_BandEnergyGetQ31	Get band energy estimate for a given filter band. "Q31" suffix designates this routine is for signals with Q31 fixed point format.

7) Fixed Point Typedefs

	Name	Description
	libq_q0d15_t	Typedef for the Q0.15 fixed point data type.
	libq_q15_t	Typedef for the Q0.15 fixed point data type.
	libq_q0d16_t	Typedef for the Q0.16 fixed point data type.
	libq_q0d31_t	Typedef for the Q0.31 fixed point data type.
	libq_q31_t	Typedef for the Q0.31 fixed point data type.
	libq_q0d63_t	Typedef for the Q0.63 fixed point data type
	libq_q63_t	Typedef for the Q0.63 fixed point data type
	libq_q15d16_t	Typedef for the Q15.16 fixed point data type
	libq_q16d15_t	Typedef for the Q16d15 fixed point data type

8) Data Types and Constants

	Name	Description
	AUDIO_EQUALIZER_MAX_NBANDS	Maximum number of filter bands supported.
	BAND_ENERGY_UNITS	Determines what units are used in reporting band energy.
	EQUALIZER_FILTER	Typedef for equalizer IIR filter definition structure.
	EQUALIZER_FILTER_32	Typedef for equalizer IIR filter definition structure.
	EQUALIZER_FILTER_GAIN	Typedef for equalizer filter gain structure.
	EQUALIZER_FILTER_GAIN_32	Typedef for equalizer filter gain structure.
	HALF_L1_TO_L2_FACTOR	Converts L1 norm (average absolute value) to L2 norm (RMS).

Description

This section describes the Application Programming Interface (API) functions of the Audio Equalizer Filtering [library](#)

1.9.1 1) Filter Routines In C

1.9.1.1 AUDIO_EQUALIZER_IIRinQ15andC Function

C

```
libq_q15_t AUDIO_EQUALIZER_IIRinQ15andC(
    EQUALIZER_FILTER * pFilter,
    bool bApplyGain,
    libq_q15_t Xin
);
```

Description

Applies equalization filter defined by *pFilter to Xin and provides single output. Optionally applies total filter gain (bApplyGain == true) or returns filter output for unity gain (bApplyGain == false). Routine is coded in C and is intended to be a testbed for an assembly versions of the algorithm.

Preconditions

The delay register values in the structure specified by pFilter should be initialized to zero prior to the first call to the function, they are updated during each filter pass.

Parameters

Parameters	Description
pFilter	pointer to filter definition structure
bApplyGain	if true applies total filter gain to output, if false applies only unity gain.
Xin	Q15 input to filter

Returns

Yout - Filter output, as Q15 fixed point.

Remarks

None.

Example

```
int16_t Xin, Yout;
EQUALIZER_FILTER myFilter = { FilterGoesHere };

Yout = AUDIO_EQUALIZER_IIRinQ15andC( &myFilter, true, Xin );
```

Or you may apply gain after getting Yout:

```
libq_q15_t Xin, Yout;
libq_q31_t Y32;
EQUALIZER_FILTER myFilter = { FilterGoesHere };

Yout = AUDIO_EQUALIZER_IIRinQ15andC( &myFilter, false, Xin );
Y32 = (myFilter.G.fracGain*Yout)<<(myFilter.G.expGain+1);
Yout = Y32>>16;
```

1.9.1.2 AUDIO_EQUALIZER_IIRinQ15FastC Function

C

```
libq_q15_t AUDIO_EQUALIZER_IIRinQ15FastC(
    EQUALIZER_FILTER * pFilter,
    libq_q15_t Xin
);
```

Description

Applies equalization filter defined by *pFilter to Xin and provides single output. This routine is designed to be faster than [AUDIO_EQUALIZER_IIRinQ15andC](#). It does not support applying the filter's gain internally.

Preconditions

The delay register values in the structure specified by pFilter should be initialized to zero prior to the first call to the function, they are updated during each filter pass.

Parameters

Parameters	Description
pFilter	pointer to filter definition structure
Xin	Q15 input to filter

Returns

Yout - Filter output, as Q15 fixed point.

Remarks

None.

Example

```
libq_q15_t Xin, Yout;
libq_q31_t Y32;
EQUALIZER_FILTER myFilter = { FilterGoesHere };

Yout = AUDIO_EQUALIZER_IIRinQ15FastC( &myFilter, Xin );
Y32 = (myFilter.G.fracGain*Yout)<<(myFilter.G.expGain+1);
Yout = Y32>>16;
```

1.9.1.3 AUDIO_EQUALIZER_IIRinQ31andC Function

C

```
libq_q31_t AUDIO_EQUALIZER_IIRinQ31andC(
    EQUALIZER_FILTER_32 * pFilter,
    bool bApplyGain,
    libq_q31_t Xint
);
```

Description

Applies equalization filter defined by *pFilter to Xin and provides single output. Optionally applies total filter gain (bApplyGain == true) or returns filter output for unity gain (bApplyGain == false). Routine is coded in C and is intended to be a testbed for an assembly versions of the algorithm.

Preconditions

The delay register values in the structure specified by pFilter should be initialized to zero prior to the first call to the function, they are updated during each filter pass.

Parameters

Parameters	Description
pFilter	pointer to filter definition structure
bApplyGain	if true applies total filter gain to output, if false applies only unity gain.
Xin	Q31 input to filter

Returns

Yout - Filter output, as Q31 fixed point.

Remarks

TO BE DONE.

Example

```
libq_q31_t Xin,Yout;
EQUALIZER_FILTER_32 myFilter = { FilterGoesHere };

Yout = AUDIO_EQUALIZER_IIRinQ31andC( &myFilter, true, Xin );
```

Or you may apply gain after getting Yout:

```
libq_q31_t Xin,Yout,Ytemp;
libq_q63_t Y64;
EQUALIZER_FILTER_32 myFilter = { FilterGoesHere };

Ytemp = AUDIO_EQUALIZER_IIRinQ31andC( &myFilter, false, Xin );
Y64 = ( myFilter.G.fracGain*((libq_q63_t)Ytemp) )<<(myFilter.G.expGain+1);
// fracGain*Ytemp is 32 bits * 32 bits = 64 bits, so must cast into 64 bits
Yout = Y64>>32; // 32 bits (inside of 64) shifted to fit into 32 bits
```

1.9.2 2) Filter Routines In Assembly

1.9.2.1 AUDIO_EQUALIZER_IIRinQ15 Function

C

```
libq_q15_t AUDIO_EQUALIZER_IIRinQ15(
    EQUALIZER_FILTER * pFilter,
    libq_q15_t Xin
);
```

Description

Applies equalization filter defined by *pFilter to Xin and provides single output. This routine is coded in MIPS assembly for maximum efficiency. It is the signal processing equivalent of [AUDIO_EQUALIZER_IIRinQ15andC](#) with bApplyGain set to false.

Calculates a single pass IIR biquad filter on Xin, and delivers the result as a 16-bit output. All math is performed using 32 bit instructions, which results truncated to 16-bits for the output. The delay register is stored as a 32-bit value for subsequent calls.

The biquad has the form:

$$Y = X(0)*b_0 + (b_1 * X(-1)) + (b_2 * X(-2)) - (a_1 * Y(-1)) - (a_2 * Y(-2))$$

All values are fractional Q15 and Q31, see data structure for specifics.

Preconditions

The delay register values in the structure specified by pFilter should be initialized to zero prior to the first call to the function, they are updated during each filter pass.

Parameters

Parameters	Description
pFilter	pointer to filter definition structure
Xin	Q15 input to filter

Returns

Yout - Filter output, as Q15 fixed point.

Remarks

This is the assembly-coded version of [AUDIO_EQUALIZER_IIRinQ15andC](#) except that it does not apply filter gain (bApplyGain == false)

An Alpha value of 2 (log2Alpha=1) has been hard coded into the function. This implies that all coefficients should be input at half value. This guarantees that all coefficients can be represented as Q15 fixed point.

If you are implementing more than one biquad IIR see [AUDIO_EQUALIZER_CascadeinQ15](#) and [AUDIO_EQUALIZER_ParallelinQ15](#)

Example

```
libq_q15_t Xin,Yout;
libq_q31_t Y32;
EQUALIZER_FILTER myFilter = { FilterGoesHere };

Yout = AUDIO_EQUALIZER_IIRinQ15( &myFilter, Xin );
Y32 = (myFilter.G.fracGain*Yout)<<(myFilter.G.expGain+1);
Yout = Y32>>16;
```

File Name

audio_eq_iir_q15.s

1.9.2.2 AUDIO_EQUALIZER_IIRinQ31 Function

C

```
libq_q31_t AUDIO_EQUALIZER_IIRinQ31(
    EQUALIZER_FILTER_32 * pFilter,
    libq_q31_t Xin
);
```

Description

Applies equalization filter defined by *pFilter to Xin and provides single output. This routine is coded in MIPS assembly for maximum efficiency. It is the signal processing equivalent of [AUDIO_EQUALIZER_IIRinQ31andC](#) with bApplyGain set to false.

Calculates a single pass IIR biquad filter on Xin, and delivers the result as a 32-bit output. All math is performed using 32 bit instructions, which results truncated to 32-bits for the output. The delay register is stored as a 32-bit value for subsequent calls. All values are fractional Q31.

The biquad has the form:

$$Y = X(0)*b0 + (b1 * X(-1)) + (b2 * X(-2)) - (a1 * Y(-1)) - (a2 * Y(-2))$$

Preconditions

The delay register values in the structure specified by pFilter should be initialized to zero prior to the first call to the function, they are updated during each filter pass.

Parameters

Parameters	Description
pFilter	pointer to filter definition structure
Xin	Q31 input to filter

Returns

Yout - Filter output, as Q31 fixed point.

Remarks

An Alpha value of 2 (log2Alpha=1) has been hard coded into the function. This implies that all coefficients should be input at half value. This guarantees that all coefficients can be represented as Q15 fixed point.

Example

```
libq_q31_t Xin,Yout;
libq_q63_t Y64;
EQUALIZER_FILTER_32 myFilter = { FilterGoesHere };

Yout = AUDIO_EQUALIZER_IIRinQ31( &myFilter, false, Xin );
Y64 = ( myFilter.G.fracGain*((libq_q63_t)Yout) )<<(myFilter.G.expGain+1);
// fracGain*Yout is 32 bits * 32 bits = 64 bits, so must cast Yout into 64 bits
Yout = Y64>>32; // 32 bits (inside of 64) shifted to fit into 32 bits
```

File Name

audio_eq_iir_q31.s

1.9.3 3) Single Band, Cascade of IIRs, in Assembly

1.9.3.1 AUDIO_EQUALIZER_Cascade2inQ15 Function

C

```
libq_q15_t AUDIO_EQUALIZER_Cascade2inQ15(
    EQUALIZER_FILTER * pFilter_Array,
    libq_q15_t Xin
);
```

Description

Function5: `libq_q15_t` AUDIO_EQUALIZER_Cascade2inQ15(`EQUALIZER_FILTER` *pFilter_Array, `libq_q15_t` Xin);

Calculates a single output of a cascade of 8 biquad IIR filters based on a single 16-bit input, and delivers the result as a 16-bit output. The cascade of filters consists of 2 separate biquad filters arranged in series such that the output of one is provided as the input to the next.

Each biquad has the form:

$$Y = X(0)*b_0 + (b_1 * X(-1)) + (b_2 * X(-2)) - (a_1 * Y(-1)) - (a_2 * Y(-2))$$

Separate filter coefficients and delay lines are provided for each of the 2 biquads in the cascade. Filter coefficients are stored as Q15 and the delay lines are 32 bits wide. Filter memory between calls is maintained in the delay lines.

Gain values (fracGain and expGain) for the first filter in the cascade is ignored. (`AUDIO_EQUALIZER_GainNormalizeQ15` sets these gains to unity.) Only the gain of the last filter is applied the cascade's output.

```
Xin -->Filter[0]-->Filter[1]-->(x)-->(<<)---> Yout
          fracGain[1]-^          ^-expGain[1]
```

Preconditions

The pFilter_Array must contain 2 [EQUALIZER_FILTER](#) elements.

The delay register values in the structure specified by pFilter_Array should be initialized to zero prior to the first call to the function, they are updated during each filter pass.

Parameters

Parameters	Description
pFilter_Array	pointer to filter coef and delay array structure
Xin	input data element X (libq_q15_t)

Returns

Sample output Y ([libq_q15_t](#))

Remarks

All delay registers values should be initialized to zero prior to the first call to the function, they are updated each pass.

An Alpha value of 2 ($\log_2 \text{Alpha} = 1$) has been hard coded into the function. This implies that all coefficients should be input at half value. This guarantees that all coefficients can be represented as Q15 fixed point.

Values for fracGain and expGain are stored in the last filter of the structure. All other fracGain and expGain values are ignored.

The only functional difference between [AUDIO_EQUALIZER_Cascade2inQ15](#) and [AUDIO_EQUALIZER_Cascade8inQ15](#), besides the obvious difference in names, is found in a single line of assembly:

```
<    addu    $s6, $a0, 48      # end address is 2*24 (len of filt stucture)
---
>    addu    $s6, $a0, 192    # end address is 8*24 (len of filt stucture)
```

So support for any number of cascaded filters is possible by simply cloning [AUDIO_EQUALIZER_Cascade2inQ15](#) into [AUDIO_EQUALIZER_CascadeinQ15](#) and changing the constant from 48 to $N \times 24$.

Example

```
libq_q15_t Xin,,Yout;
EQUALIZER_FILTER myFilterArray[ ] = { FILTER1, FILTER2 };

// Normalize filter gains
AUDIO_EQUALIZER_GainNormalizeQ15( 1, 2, myFilterArray );
// Gains for for the first filter is set to {0.5,1}, but is ignored.
// Gain for 8th filter is applied after call to calculate filter cascade.

while ( bGotData )
{
    // Get Xin;
    // Filter
    Yout = AUDIO_EQUALIZER_Cascade2inQ15( myFilterArray, Xin );
    // Play Yout;
}
```

File Name

audio_eq_cascade2_q15.s

1.9.3.2 AUDIO_EQUALIZER_Cascade2inQ31 Function

C

```
libq_q31_t AUDIO_EQUALIZER_Cascade2inQ31(
    EQUALIZER_FILTER_32 * pFilter_Array,
```

```
libq_q31_t Xin
);
```

Description

Calculates a single output of a cascade of 2 biquad IIR filters based on a single 16-bit input, and delivers the result as a 16-bit output. The cascade of filters consists of 2 separate biquad filters arranged in series such that the output of one is provided as the input to the next.

Each biquad has the form:

$$Y = X(0)*b0 + (b1 * X(-1)) + (b2 * X(-2)) - (a1 * Y(-1)) - (a2 * Y(-2))$$

Separate filter coefficients and delay lines are provided for each of the 2 biquads in the cascade. Filter coefficients are stored as Q15 and the delay lines are 32 bits wide. Filter memory between calls is maintained in the delay lines.

Gain values (fracGain and expGain) for the first 7 filters in the cascade are ignored. ([AUDIO_EQUALIZER_GainNormalizeQ15](#) sets these gains to unity.) Only the gain of the last filter is applied the cascade's output.

```
Xin -->Filter[0]-->Filter[1]-->(x)-->(<<)---> Yout
      fracGain[1]-^      ^-expGain[1]
```

Preconditions

pFilter_Array must contain 2 [EQUALIZER_FILTER_32](#) elements.

The delay register values in the structure specified by pFilter_Array should be initialized to zero prior to the first call to the function, they are updated during each filter pass.

Parameters

Parameters	Description
pFilter_Array	pointer to filter coef and delay array structure
Xin	input data element X (libq_q15_t)

Returns

Sample output Y ([libq_q15_t](#))

Remarks

All delay registers values should be initialized to zero prior to the first call to the function, they are updated each pass.

An Alpha value of 2 (log2Alpha=1) has been hard coded into the function. This implies that all coefficients should be input at half value. This guarantees that all coefficients can be represented as Q15 fixed point.

Values for fracGain and expGain are stored in the last filter of the structure. All other fracGain and expGain values are ignored.

The only functional difference between [AUDIO_EQUALIZER_Cascade2inQ31](#) and [AUDIO_EQUALIZER_Cascade8inQ31](#), besides the obvious difference in names, is found in a single line of assembly:

```
<    addu    $s6, $a0, 80          # end address is 2*40 (len of filt stucture)
---
>    addu    $s6, $a0, 320        # end address is 8*40 (len of filt stucture)
```

So support for any number of cascaded filters is possible by simply cloning [AUDIO_EQUALIZER_Cascade2inQ31](#) into [AUDIO_EQUALIZER_CascadeinQ31](#) and changing the constant from 80 to N*40.

Example

```
libq_q31_t Xin,,Yout;
libq_q31_t Y32;
EQUALIZER_FILTER_32 myFilterArray[ ] = { FILTER1, FILTER2 };

// Normalize filter gains
AUDIO_EQUALIZER_GainNormalizeQ31( 1, 2, myFilterArray );
// Gains for for the first filters is set to {0.5,1}, but are ignored.
// Gain for 8th filter is applied after call to calculate filter cascade.
```



```

while ( bGotData )
{
    // Get Xin
    // Filter
    Yout = AUDIO_EQUALIZER_Cascade2inQ31( myFilterArray, Xin );
    // Play Yout;
}

```

File Name

audio_eq_cascade2_q31.s

1.9.3.3 AUDIO_EQUALIZER_Cascade8inQ15 Function

C

```

libq_q15_t AUDIO_EQUALIZER_Cascade8inQ15(
    EQUALIZER_FILTER * pFilter_Array,
    libq_q15_t Xin
);

```

Description

Function5: `libq_q15_t AUDIO_EQUALIZER_Cascade8inQ15(EQUALIZER_FILTER *pFilter_Array, libq_q15_t Xin);`

Calculates a single output of a cascade of 8 biquad IIR filters based on a single 16-bit input, and delivers the result as a 16-bit output. The cascade of filters consists of 8 separate biquad filters arranged in series such that the output of one is provided as the input to the next.

Each biquad has the form:

$$Y = X(0)*b_0 + (b_1 * X(-1)) + (b_2 * X(-2)) - (a_1 * Y(-1)) - (a_2 * Y(-2))$$

Separate filter coefficients and delay lines are provided for each of the 8 biquads in the cascade. Filter coefficients are stored as Q15 and the delay lines are 32 bits wide. Filter memory between calls is maintained in the delay lines.

Gain values (fracGain and expGain) for the first 7 filters in the cascade are ignored. (`AUDIO_EQUALIZER_GainNormalizeQ15` sets these gains to unity.) Only the gain of the last filter is applied the cascade's output.

```

Xin -->Filter[0]-->Filter[1]-->Filter[2]-->Filter[3]--+
|
+-----+
|
+-->Filter[4]-->Filter[5]-->Filter[6]-->Filter[7]-->(x)-->(<<)---> Yout
                                fracGain[7]-^      ^-expGain[7]

```

Preconditions

pFilter_Array must contain 8 `EQUALIZER_FILTER` elements.

The delay register values in the structure specified by pFilter_Array should be initialized to zero prior to the first call to the function, they are updated during each filter pass.

Parameters

Parameters	Description
pFilter_Array	pointer to filter coef and delay array structure
Xin	input data element X (<code>libq_q15_t</code>)

Returns

Sample output Y (`libq_q15_t`)

Remarks

All delay registers values should be initialized to zero prior to the first call to the function, they are updated each pass.

An Alpha value of 2 ($\log_2 \text{Alpha}=1$) has been hard coded into the function. This implies that all coefficients should be input at half value. This guarantees that all coefficients can be represented as Q15 fixed point.

Values for fracGain and expGain are stored in the last filter of the structure. All other fracGain and expGain values are ignored.

The only functional difference between `AUDIO_EQUALIZER_Cascade8inQ15` and `AUDIO_EQUALIZER_Cascade2inQ15`, besides the obvious difference in names, is found in a single line of assembly:

```
<      addu    $s6, $a0, 192      # end address is 8*24 (len of filt stucture)
---
>      addu    $s6, $a0, 48       # end address is 2*24 (len of filt stucture)
```

So support for any number of cascaded filters is possible by simply cloning `AUDIO_EQUALIZER_Cascade8inQ15` into `AUDIO_EQUALIZER_CascadeinQ15` and changing the constant from 192 to $N*24$.

Example

```
libq_q15_t Xin,,Yout;
EQUALIZER_FILTER myFilterArray[ ] = { FILTER1, FILTER2, FILTER3, FILTER4,
                                       FILTER5, FILTER6, FILTER7, FILTER8 };

// Normalize filter gains
AUDIO_EQUALIZER_GainNormalizeQ15( 1, 8, myFilterArray );
// Gains for first 7 filters set to {0.5,1}, but are ignored.
// Gain for 8th filter is applied after call to calculate filter cascade.

while ( bGotData )
{
    // Get Xin;
    // Filter
    Yout = AUDIO_EQUALIZER_Cascade8inQ15( myFilterArray, Xin );
    // Play Yout;
}
```

File Name

audio_eq_cascade8_q15.s

1.9.3.4 AUDIO_EQUALIZER_Cascade8inQ31 Function

C

```
libq_q31_t AUDIO_EQUALIZER_Cascade8inQ31(
    EQUALIZER_FILTER_32 * pFilter_Array,
    libq_q31_t Xin
);
```

Description

Calculates a single output of a cascade of 8 biquad IIR filters based on a single 16-bit input, and delivers the result as a 16-bit output. The cascade of filters consists of 8 separate biquad filters arranged in series such that the output of one is provided as the input to the next.

Each biquad has the form:

$$Y = X(0)*b_0 + (b_1 * X(-1)) + (b_2 * X(-2)) - (a_1 * Y(-1)) - (a_2 * Y(-2))$$

Separate filter coefficients and delay lines are provided for each of the 8 biquads in the cascade. Filter coefficients are stored as Q15 and the delay lines are 32 bits wide. Filter memory between calls is maintained in the delay lines.

Gain values (fracGain and expGain) for the first 7 filters in the cascade are ignored. ([AUDIO_EQUALIZER_GainNormalizeQ15](#)

sets these gains to unity.) Only the gain of the last filter is applied the cascade's output.

```

Xin -->Filter[0]-->Filter[1]-->Filter[2]-->Filter[3]--+
|
+-----+
|
+-->Filter[4]-->Filter[5]-->Filter[6]-->Filter[7]--> (x) --> (<<) ---> Yout
                    fracGain[7]-^          ^-expGain[7]

```

Preconditions

pFilter_Array must contain 8 [EQUALIZER_FILTER_32](#) elements.

The delay register values in the structure specified by pFilter_Array should be initialized to zero prior to the first call to the function, they are updated during each filter pass.

Parameters

Parameters	Description
pFilter_Array	pointer to filter coef and delay array structure
Xin	input data element X (libq_q15_t)

Returns

Sample output Y ([libq_q15_t](#))

Remarks

All delay registers values should be initialized to zero prior to the first call to the function, they are updated each pass.

An Alpha value of 2 (log2Alpha=1) has been hard coded into the function. This implies that all coefficients should be input at half value. This guarantees that all coefficients can be represented as Q15 fixed point.

Values for fracGain and expGain are stored in the last filter of the structure. All other fracGain and expGain values are ignored.

The only functional difference between [AUDIO_EQUALIZER_Cascade8inQ31](#) and [AUDIO_EQUALIZER_Cascade2inQ31](#), besides the obvious difference in names, is found in a single line of assembly:

```

<    addu    $s6, $a0, 320      # end address is 8*40 (len of filt stucture)
---
>    addu    $s6, $a0, 80       # end address is 2*40 (len of filt stucture)

```

So support for any number of cascaded filters is possible by simply cloning [AUDIO_EQUALIZER_Cascade8inQ31](#) into [AUDIO_EQUALIZER_CascadeinQ31](#) and changing the constant from 320 to N*40.

Example

```

libq_q31_t Xin,,Yout;
libq_q31_t Y32;
EQUALIZER_FILTER_32 myFilterArray[ ] = { FILTER1, FILTER2, FILTER3, FILTER4,
                                           FILTER5, FILTER6, FILTER7, FILTER8 };

// Normalize filter gains
AUDIO_EQUALIZER_GainNormalizeQ31( 1, 8, myFilterArray );
// Gains for first 7 filters set to {0.5,1}, but are ignored.
// Gain for 8th filter is applied after call to calculate filter cascade.

while ( bGotData )
{
    // Get Xin
    // Filter
    Yout = AUDIO_EQUALIZER_Cascade8inQ31( myFilterArray, Xin );
    // Play Yout;
}

```

File Name

audio_eq_cascade8_q31.s

1.9.4 4) Multiple Bands, Multiple Filters per Band, in Assembly

1.9.4.1 AUDIO_EQUALIZER_Parallel4x2inQ15 Function

C

```
libq_q15_t AUDIO_EQUALIZER_Parallel4x2inQ15(
    EQUALIZER_FILTER * pFilter_Array,
    libq_q15_t Xin
);
```

Description

Calculates 4 parallel IIR filters on Xin, sums the result and delivers the result as a 16-bit output. Each parallel filter is a cascade of 2 biquad IIR filters in series each, for a total of 8 filters calculated.

Each biquad has the form:

$$Y = X(0)*b0 + (b1 * X(-1)) + (b2 * X(-2)) - (a1 * Y(-1)) - (a2 * Y(-2))$$

All math is performed using 32 bit instructions, with results truncated to 16-bits for the output. Delay registers are stored as a 32-bit value for subsequent calls. All values are fractional Q15 and Q31, see data structure for specifics.

The filter structure has 4 parallel bands, each band with 2 filters:

```
Xin ---+-->Filter[0]-->Filter[1]-->(x)----+
      |          fracGain[1]-^          V
      +-->Filter[2]-->Filter[3]-->(x)--->(+)---+
      |          fracGain[3]-^          V
      +-->Filter[4]-->Filter[5]-->(x)----->(+)---+
      |          fracGain[5]-^          V
      +-->Filter[6]-->Filter[7]-->(x)----->(+)--->(<<)----> Yout
      |          fracGain[7]-^          expGain[7]-^
```

Output is tuned by 2 multiplier factors. First each parallel section has a fractional gain (attenuation) that enables individual scaling of that section. A global binary (log2N) gain is applied to the final result. The combination of these gain factors enable both gain and attenuation.

See [AUDIO_EQUALIZER_GainNormalizeQ15](#) for information on gain normalization.

Preconditions

Delay register values should be initialized to zero. pFilter_Array must contain 4x2 = 8 [EQUALIZER_FILTER](#) elements.

Parameters

Parameters	Description
pFilter_Array	pointer to filter coef and delay array structure
Xin	input data element X (libq_q15_t)

Returns

Sample output Y ([libq_q15_t](#))

Remarks

Requires 4*2 filter structures initialized with coefficients pointed to in an array, pFilterArray[0] and pFilterArray[1] apply to the first parallel filter segment. [2] and [3] apply to the second segment and so forth.

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass.

An Alpha value of 2 ($\log_2 \text{Alpha}=1$) has been hard coded into the function. This implies that all coefficients should be input at half value. This guarantees that all coefficients can be represented as Q15 fixed point.

A div 4 has been applied in the function for each block, to normalize the 4 blocks are summed to form the end output. Each block has a functional hard-coded gain of 0.25.

Digital attenuation on a per parallel channel basis has been applied by fracgain. This is a Q15 fractional value, and is passed in from the second series filter of that parallel structure.

Binary gain globally has been applied to the output sum of the function, this is received in the last filter coef block of the function.

A number of similar filters exist that enable the user to trade-off processing resources (cycles) and filter design format. There are variations of each type for 16-bit and 32-bit data. Generally the 16-bit functions are mathematically less complex, and take about 25% fewer processor cycles than the 32-bit versions.

Parallel filters may have a fixed or variable number of parallel elements - 4, 8 or N. As the number is higher the processor cycles increase, and the general N form requires about 8% more processing cycles than its fixed value counterpart.

Similarly, the number of cascade (in series) biquad filter blocks per parallel filter section (M) can be fixed at 2, or a variable. The variable version requires about 2% more processing power than the fixed equivalent.

When possible on a processing-limited embedded system, it is advised to use the fixed versions where possible, although more general versions are available to present the widest variety of design choices.

Example

```
libq_q15_t Xin,Yout;
EQUALIZER_FILTER_32 myFilterArray[ ] = { FILTER1, FILTER2, FILTER3, FILTER4,
                                           FILTER5, FILTER6, FILTER7, FILTER8 };

// Normalize filter gains
AUDIO_EQUALIZER_GainNormalizeQ15( 4, 2, myFilterArray );

while ( bGotData )
{
    // Get Xin, filter it
    Yout = AUDIO_EQUALIZER_Parallel4x2inQ15( myFilterArray, Xin );
    // Play Yout
}
```

File Name

audio_eq_parallel4x2_q15.s

1.9.4.2 AUDIO_EQUALIZER_Parallel4x2inQ31 Function

C

```
libq_q31_t AUDIO_EQUALIZER_Parallel4x2inQ31(
    EQUALIZER_FILTER_32 * pFilter_Array,
    libq_q31_t Xin
);
```

Description

Calculates 4 parallel IIR filters on Xin, sums the result and delivers the result as a 32-bit output. Each parallel filter is a cascade of 2 biquad IIR filters in series each, for a total of 8 filters calculated.

Each biquad has the form:

$$Y = X(0)*b_0 + (b_1 * X(-1)) + (b_2 * X(-2)) - (a_1 * Y(-1)) - (a_2 * Y(-2))$$

All math is performed using 32 bit instructions, with results truncated to 32-bits for the output. The delay register is stored as a

32-bit value for subsequent calls. All values are fractional Q31.

The filter structure has 4 parallel bands, each band with 2 filters:

```

Xin ---->Filter[0]-->Filter[1]-->(x)----+
      |               fracGain[1]-^      V
      +-->Filter[2]-->Filter[3]-->(x)-->(+)--+
      |               fracGain[3]-^      V
      +-->Filter[4]-->Filter[5]-->(x)----->(+)--+
      |               fracGain[5]-^      V
      +-->Filter[6]-->Filter[7]-->(x)----->(+)---->(<<)----> Yout
                        fracGain[7]-^      expGain[7]-^

```

Output is tuned by 2 multiplier factors. First each parallel section has a fractional gain (attenuation) that enables individual scaling of that section. A global binary (log2N) gain is applied to the final result. The combination of these gain factors enable both gain and attenuation.

See [AUDIO_EQUALIZER_GainNormalizeQ31](#) for information on gain normalization.

Preconditions

Delay register values should be initialized to zero. pFilter_Array must contain $4 \times 2 = 8$ [EQUALIZER_FILTER_32](#) elements

Parameters

Parameters	Description
pFilter_Array	pointer to filter coef and delay array structure
Xin	input data element X (libq_q31_t)

Returns

Sample output Y ([libq_q31_t](#))

Remarks

Requires 4×2 filter structures initialized with coefficients pointed to in an array, pFilterArray[0] and pFilterArray[1] apply to the first parallel filter segment. [2] and [3] apply to the second segment and so forth.

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass.

A coefficient gain of 2 has been hard coded into the biquad functional block. This implies that all coefs should be input at half value. This is purposeful, since many filter designs need a div2 to have each coef between the required -1

Example

```

libq_q31_t Xin,Yout;
EQUALIZER_FILTER_32 myFilterArray[ ] = { FILTER1, FILTER2, FILTER3, FILTER4,
                                           FILTER5, FILTER6, FILTER7, FILTER8 };

// Normalize filter gains
AUDIO_EQUALIZER_GainNormalizeQ31( 4, 2, myFilterArray );

while ( bGotData )
{
    // Get Xin, filter it
    Yout = AUDIO_EQUALIZER_Parallel4x2inQ31( myFilterArray, Xin );
    // Play Yout
}

```

File Name

audio_eq_parallel4x2_q31.s

1.9.4.3 AUDIO_EQUALIZER_Parallel8x2inQ15 Function

C

```
libq_q15_t AUDIO_EQUALIZER_Parallel8x2inQ15(
    EQUALIZER_FILTER * pFilter_Array,
    libq_q15_t Xin
);
```

Description

Calculates 8 parallel IIR filters on Xin, sums the result and delivers the result as a 16-bit output. Each parallel filter is a cascade of 2 biquad IIR filters in series each, for a total of 8 filters calculated.

Each biquad has the form:

$$Y = X(0)*b0 + (b1 * X(-1)) + (b2 * X(-2)) - (a1 * Y(-1)) - (a2 * Y(-2))$$

All math is performed using 32 bit instructions, with results truncated to 16-bits for the output. Delay registers are stored as a 32-bit value for subsequent calls. All values are fractional Q15 and Q31, see data structure for specifics.

The filter structure has 8 parallel bands, each band with 2 filters:

```
Xin ---->Filter[0]-->Filter[1]-->(x)----+
|          fracGain[1]-^          V
+-->Filter[2]-->Filter[3]-->(x) --> (+) --+
|          fracGain[3]-^          V
+-->Filter[4]-->Filter[5]-->(x) -----> (+) --+
|          fracGain[5]-^          V
+-->Filter[6]-->Filter[7]-->(x) -----> (+) --+
|          fracGain[7]-^          V
+-->Filter[8]-->Filter[9]-->(x) -----> (+) --+
|          fracGain[9]-^          V
+-->Filter[10]-->Filter[11]-->(x) -----> (+) --+
|          fracGain[11]-^          V
+-->Filter[12]-->Filter[13]-->(x) -----> (+) --+
|          fracGain[13]-^          V
+-->Filter[14]-->Filter[15]-->(x) -----> (+) --> (<<) --> Yout
|          fracGain[15]-^          expGain[7]-^
```

Output is tuned by 2 multiplier factors. First each parallel section has a fractional gain (attenuation) that enables individual scaling of that section. A global binary (log2N) gain is applied to the final result. The combination of these gain factors enable both gain and attenuation.

See [AUDIO_EQUALIZER_GainNormalizeQ15](#) for information on gain normalization.

Preconditions

Delay register values should be initialized to zero. pFilter_Array must contain 8x2 = 16 [EQUALIZER_FILTER](#) elements.

Parameters

Parameters	Description
pFilter_Array	pointer to filter coef and delay array structure
Xin	input data element X (libq_q15_t)

Returns

Sample output Y ([libq_q15_t](#))

Remarks

Requires 8*2 filter structures initialized with coefficients pointed to in an array, pFilterArray[0] and pFilterArray[1] apply to the first parallel filter segment. [2] and [3] apply to the second segment and so forth.

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass.

An Alpha value of 2 ($\log_2 \text{Alpha}=1$) has been hard coded into the function. This implies that all coefficients should be input at half value. This guarantees that all coefficients can be represented as Q15 fixed point.

A div 8 has been applied in the function for each block, to normalize the 4 blocks are summed to form the end output. Each block has a functional hard-coded gain of 0.125.

Digital attenuation on a per parallel channel basis has been applied by fracgain. This is a Q15 fractional value, and is passed in from the second series filter of that parallel structure.

Binary gain globally has been applied to the output sum of the function, this is received in the last filter coef block of the function.

A number of similar filters exist that enable the user to trade-off processing resources (cycles) and filter design format. There are variations of each type for 16-bit and 32-bit data. Generally the 16-bit functions are mathematically less complex, and take about 25% fewer processor cycles than the 32-bit versions.

Parallel filters may have a fixed or variable number of parallel elements - 4, 8 or N. As the number is higher the processor cycles increase, and the general N form requires about 8% more processing cycles than its fixed value counterpart.

Similarly, the number of cascade (in series) biquad filter blocks per parallel filter section (M) can be fixed at 2, or a variable. The variable version requires about 2% more processing power than the fixed equivalent.

When possible on a processing-limited embedded system, it is advised to use the fixed versions where possible, although more general versions are available to present the widest variety of design choices.

Example

```
libq_q15_t Xin,Yout;
EQUALIZER_FILTER myFilterArray[ ] = { BAND1_FILTER1, BAND1_FILTER2,
                                       BAND2_FILTER1, BAND2_FILTER2,
                                       BAND3_FILTER1, BAND3_FILTER2,
                                       BAND4_FILTER1, BAND4_FILTER2,
                                       BAND5_FILTER1, BAND5_FILTER2,
                                       BAND6_FILTER1, BAND6_FILTER2,
                                       BAND7_FILTER1, BAND7_FILTER2,
                                       BAND8_FILTER1, BAND8_FILTER2 };

// Normalize filter gains
AUDIO_EQUALIZER_GainNormalizeQ15( 8, 2, myFilterArray );

while ( bGotData )
{
    // Get Xin, filter it
    Yout = AUDIO_EQUALIZER_Parallel8x2inQ15( myFilterArray, Xin );
    // Play Yout
}
```

File Name

audio_eq_parallel8x2_q15.s

1.9.4.4 AUDIO_EQUALIZER_Parallel8x2inQ31 Function

C

```
libq_q31_t AUDIO_EQUALIZER_Parallel8x2inQ31(
    EQUALIZER_FILTER * pFilter_Array,
    libq_q31_t Xin
);
```

Description

Calculates 8 parallel IIR filters on Xin, sums the result and delivers the result as a 32-bit output. Each parallel filter is a cascade

of 2 biquad IIR filters in series each, for a total of 8 filters calculated.

Each biquad has the form:

$$Y = X(0)*b0 + (b1 * X(-1)) + (b2 * X(-2)) - (a1 * Y(-1)) - (a2 * Y(-2))$$

All math is performed using 32 bit instructions, with results truncated to 16-bits for the output. Delay registers are stored as a 32-bit value for subsequent calls. All values are fractional Q31, see data structure for specifics.

The filter structure has 8 parallel bands, each band with 2 filters:

```

Xin  --->Filter[0]-->Filter[1]-->(x)----+
      |          fracGain[1]-^          V
      +--->Filter[2]-->Filter[3]-->(x)--->(+)--+
      |          fracGain[3]-^          V
      +--->Filter[4]-->Filter[5]-->(x)----->(+)--+
      |          fracGain[5]-^          V
      +--->Filter[6]-->Filter[7]-->(x)----->(+)--+
      |          fracGain[7]-^          V
      +--->Filter[8]-->Filter[9]-->(x)----->(+)--+
      |          fracGain[9]-^          V
      +--->Filter[10]-->Filter[11]-->(x)----->(+)--+
      |          fracGain[11]-^          V
      +--->Filter[12]-->Filter[13]-->(x)----->(+)--+
      |          fracGain[13]-^          V
      +--->Filter[14]-->Filter[15]-->(x)----->(+)-->(<<)---> Yout
                                     fracGain[15]-^      expGain[7]-^

```

Output is tuned by 2 multiplier factors. First each parallel section has a fractional gain (attenuation) that enables individual scaling of that section. A global binary (log2N) gain is applied to the final result. The combination of these gain factors enable both gain and attenuation.

See [AUDIO_EQUALIZER_GainNormalizeQ31](#) for information on gain normalization.

Preconditions

Delay register values should be initialized to zero. pFilter_Array must contain $8 \times 2 = 16$ [EQUALIZER_FILTER_32](#) elements.

Parameters

Parameters	Description
pFilter_Array	pointer to filter coef and delay array structure
Xin	input data element X (libq_q15_t)

Returns

Sample output Y ([libq_q15_t](#))

Remarks

Requires 8×2 filter structures initialized with coefficients pointed to in an array, pFilterArray[0] and pFilterArray[1] apply to the first parallel filter segment. [2] and [3] apply to the second segment and so forth.

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass.

An Alpha value of 2 (log2Alpha=1) has been hard coded into the function. This implies that all coefficients should be input at half value. This guarantees that all coefficients can be represented as Q15 fixed point.

A div 8 has been applied in the function for each block, to normalize the 4 blocks are summed to form the end output. Each block has a functional hard-coded gain of 0.125.

Digital attenuation on a per parallel channel basis has been applied by fracgain. This is a Q15 fractional value, and is passed in from the second series filter of that parallel structure.

Binary gain globally has been applied to the output sum of the function, this is received in the last filter coef block of the function.

A number of similar filters exist that enable the user to trade-off processing resources (cycles) and filter design format. There are

variations of each type for 16-bit and 32-bit data. Generally the 16-bit functions are mathematically less complex, and take about 25% fewer processor cycles than the 32-bit versions.

Parallel filters may have a fixed or variable number of parallel elements - 4, 8 or N. As the number is higher the processor cycles increase, and the general N form requires about 8% more processing cycles than its fixed value counterpart.

Similarly, the number of cascade (in series) biquad filter blocks per parallel filter section (M) can be fixed at 2, or a variable. The variable version requires about 2% more processing power than the fixed equivalent.

When possible on a processing-limited embedded system, it is advised to use the fixed versions where possible, although more general versions are available to present the widest variety of design choices.

Example

```
libq_q31_t Xin,Yout;
EQUALIZER_FILTER_32 myFilterArray[ ] = { BAND1_FILTER1, BAND1_FILTER2,
                                           BAND2_FILTER1, BAND2_FILTER2,
                                           BAND3_FILTER1, BAND3_FILTER2,
                                           BAND4_FILTER1, BAND4_FILTER2,
                                           BAND5_FILTER1, BAND5_FILTER2,
                                           BAND6_FILTER1, BAND6_FILTER2,
                                           BAND7_FILTER1, BAND7_FILTER2,
                                           BAND8_FILTER1, BAND8_FILTER2 };

// Normalize filter gains
AUDIO_EQUALIZER_GainNormalizeQ31( 8, 2, myFilterArray );

while ( bGotData )
{
    // Get Xin, filter it
    Yout = AUDIO_EQUALIZER_Parallel8x2inQ31( myFilterArray, Xin );
    // Play Yout
}
```

File Name

audio_eq_parallel8x2_q31.s

1.9.4.5 AUDIO_EQUALIZER_ParallelNx2inQ15 Function

C

```
libq_q15_t AUDIO_EQUALIZER_ParallelNx2inQ15(
    EQUALIZER_FILTER * pFilter_Array,
    libq_q15_t Xin,
    int N,
    libq_q31_t Scale
);
```

Description

Calculates N parallel IIR filters on Xin, sums the result and delivers the result as a 16-bit output. Each parallel filter is a cascade of 2 biquad IIR filters in series each, for a total of 2N filters calculated.

Each biquad has the form:

$$Y = X(0)*b0 + (b1 * X(-1)) + (b2 * X(-2)) - (a1 * Y(-1)) - (a2 * Y(-2))$$

All math is performed using 32 bit instructions, with results truncated to 16-bits for the output. Delay registers are stored as a 32-bit value for subsequent calls. All values are fractional Q15 and Q31, see data structure for specifics.

The filter structure has N parallel bands, each band with 2 filters:

```
Xin ---->Filter[0]-->Filter[1]-->(x)----+
|                                     fracGain[1]-^      V
+-->Filter[2]-->Filter[3]-->(x)-->(+)--+
|                                     fracGain[3]-^      V
```

```

+-->Filter[4]-->Filter[5]-->(x)----->(+)--+
|          fracGain[5]-^          V
+-->Filter[6]-->Filter[7]-->(x)----->(+)--+
|          fracGain[7]-^          V
+-->Filter[8]-->Filter[9]-->(x)----->(+)--+
|          fracGain[9]-^          V
+-->Filter[10]-->Filter[11]-->(x)----->(+)--+
|          fracGain[11]-^          V
+-->Filter[12]-->Filter[13]-->(x)----->(+)--+
|          fracGain[13]-^          .
.                                     .
.                                     .
+-->Filter[2N-2]-->Filter[2N-1]-->(x)----->(+)-->(<<)--> Yout
|          fracGain[2N-1]-^          expGain[2N-1]-^

```

Output is tuned by 2 multiplier factors. First each parallel section has a fractional gain (attenuation) that enables individual scaling of that section. A global binary (log2N) gain is applied to the final result. The combination of these gain factors enable both gain and attenuation.

See [AUDIO_EQUALIZER_GainNormalizeQ15](#) for information on gain normalization.

Preconditions

Delay register values should be initialized to zero. pFilter_Array must contain $N \times 2 = 2N$ [EQUALIZER_FILTER](#) elements.

Parameters

Parameters	Description
pFilter_Array	pointer to filter coef and delay array structure
Xin	input data element X (libq_q15_t)
N	Number of parallel IIR filter elements (int)
Scale	Scaling factor on each filter element (libq_q31_t) Should be the Q0.31 equivalent of 1/N.

Returns

Sample output Y ([libq_q15_t](#))

Remarks

Requires $N \times 2$ filter structures initialized with coefficients pointed to in an array, pFilterArray[0] and pFilterArray[1] apply to the first parallel filter segment. [2] and [3] apply to the second segment and so forth.

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass.

An Alpha value of 2 (log2Alpha=1) has been hard coded into the function. This implies that all coefficients should be input at half value. This guarantees that all coefficients can be represented as Q15 fixed point. -1

Example

```

#define NBANDS 9 // number of parallel elements
libq_q31_t scaleValue = (1.0/NBANDS)*(1<<31); // Q0.31 scale value = 1/#bands

libq_q15_t Xin,Yout;
EQUALIZER_FILTER myFilterArray[ ] = { BAND1_FILTER1, BAND1_FILTER2,
                                       BAND2_FILTER1, BAND2_FILTER2,
                                       BAND3_FILTER1, BAND3_FILTER2,
                                       BAND4_FILTER1, BAND4_FILTER2,
                                       BAND5_FILTER1, BAND5_FILTER2,
                                       BAND6_FILTER1, BAND6_FILTER2,
                                       BAND7_FILTER1, BAND7_FILTER2,
                                       BAND8_FILTER1, BAND8_FILTER2,
                                       BAND9_FILTER1, BAND9_FILTER2 };

// Normalize filter gains
AUDIO_EQUALIZER_GainNormalizeQ15( NBANDS, 2, myFilterArray );

```

```

while ( bGotData )
{
    // Get Xin, filter it
    Yout = AUDIO_EQUALIZER_ParallelNx2inQ15( myFilterArray, Xin, NBANDS, scaleValue );
    // Play Yout
}

```

File Name

audio_eq_parallelNx2_q15.s

1.9.4.6 AUDIO_EQUALIZER_ParallelNx2inQ31 Function

C

```

libq_q31_t AUDIO_EQUALIZER_ParallelNx2inQ31(
    EQUALIZER_FILTER_32 * pFilter_Array,
    libq_q31_t Xin,
    int N,
    libq_q31_t Scale
);

```

Description

Calculates N parallel IIR filters on Xin, sums the result and delivers the result as a 16-bit output. Each parallel filter is a cascade of 2 biquad IIR filters in series each, for a total of 2N filters calculated.

Each biquad has the form:

$$Y = X(0)*b_0 + (b_1 * X(-1)) + (b_2 * X(-2)) - (a_1 * Y(-1)) - (a_2 * Y(-2))$$

All math is performed using 32 bit instructions. Delay registers are stored as a 32-bit value for subsequent calls. All values are fractional Q31, see data structure for specifics.

The filter structure has N parallel bands, each band with 2 filters:

```

Xin ---+--->Filter[0]-->Filter[1]-->(x)----+
|          fracGain[1]-^          V
+--->Filter[2]-->Filter[3]-->(x)--->(+)--+
|          fracGain[3]-^          V
+--->Filter[4]-->Filter[5]-->(x)----->(+)--+
|          fracGain[5]-^          V
+--->Filter[6]-->Filter[7]-->(x)----->(+)--+
|          fracGain[7]-^          V
+--->Filter[8]-->Filter[9]-->(x)----->(+)--+
|          fracGain[9]-^          V
+--->Filter[10]-->Filter[11]-->(x)----->(+)--+
|          fracGain[11]-^          V
+--->Filter[12]-->Filter[13]-->(x)----->(+)--+
|          fracGain[13]-^          V
.
.
.
+--->Filter[2N-2]-->Filter[2N-1]-->(x)----->(+)-->(<<)--> Yout
|          fracGain[2N-1]-^          expGain[2N-1]-^

```

Output is tuned by 2 multiplier factors. First each parallel section has a fractional gain (attenuation) that enables individual scaling of that section. A global binary (log2N) gain is applied to the final result. The combination of these gain factors enable both gain and attenuation.

See [AUDIO_EQUALIZER_GainNormalizeQ31](#) for information on gain normalization.

Preconditions

Delay register values should be initialized to zero. pFilter_Array must contain Nx2 = 2N [EQUALIZER_FILTER_32](#) elements.

Parameters

Parameters	Description
pFilter_Array	pointer to filter coef and delay array structure
Xin	input data element X (libq_q31_t)
N	Number of parallel IIR filter elements (int)
Scale	Scaling factor on each filter element (libq_q31_t) Should be the Q0.31 equivalent of 1/N.

Returns

Sample output Y ([libq_q31_t](#))

Remarks

Requires N*2 filter structures initialized with coefficients pointed to in an array, pFilterArray[0] and pFilterArray[1] apply to the first parallel filter segment. [2] and [3] apply to the second segment and so forth.

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass.

An Alpha value of 2 (log2Alpha=1) has been hard coded into the function. This implies that all coefficients should be input at half value. This guarantees that all coefficients can be represented as Q31 fixed point. -1

Example

```
#define NBANDS 9 // number of parallel elements
libq_q31_t scaleValue = (1.0/NBANDS)*(1<<31); // Q0.31 scale value = 1/#bands

libq_q31_t Xin,Yout;
EQUALIZER_FILTER myFilterArray[ ] = { BAND1_FILTER1, BAND1_FILTER2,
                                       BAND2_FILTER1, BAND2_FILTER2,
                                       BAND3_FILTER1, BAND3_FILTER2,
                                       BAND4_FILTER1, BAND4_FILTER2,
                                       BAND5_FILTER1, BAND5_FILTER2,
                                       BAND6_FILTER1, BAND6_FILTER2,
                                       BAND7_FILTER1, BAND7_FILTER2,
                                       BAND8_FILTER1, BAND8_FILTER2,
                                       BAND9_FILTER1, BAND9_FILTER2 };

// Normalize filter gains
AUDIO_EQUALIZER_GainNormalizeQ31( NBANDS, 2, myFilterArray );

while ( bGotData )
{
    // Get Xin, filter it
    Yout = AUDIO_EQUALIZER_ParallelNx2inQ31( myFilterArray, Xin, NBANDS, scaleValue );
    // Play Yout
}
```

File Name

audio_eq_parallelNx2_q31.s

1.9.4.7 AUDIO_EQUALIZER_ParallelNxMinQ15 Function

C

```
libq_q15_t AUDIO_EQUALIZER_ParallelNxMinQ15(
    EQUALIZER_FILTER * pFilter_Array,
    libq_q15_t Xin,
    int N,
    libq_q31_t Scale,
    int M
);
```

Description

Calculates N parallel IIR filters on Xin, sums the result and delivers the result as a 16-bit output. Each parallel filter is a cascade of M biquad IIR filters in series each, for a total of N*M filters calculated. All math is performed using 32 bit instructions, with results truncated to 16-bits for the output. The delay register is stored as a 32-bit value for subsequent functions. Output is tuned by 2 multiplier factors. First each parallel section has a fractional gain (attenuation) that enables individual scaling of that section. Second, a global binary (log2N) gain is applied to the result. The combination of gain factors enable both gain and attenuation.

To normalize the output scale is used as a constant multiply for each parallel block. This value is normally input as 1/N. This is a Q31 numerical value to maintain internal resolution.

All values are fractional Q15 and Q31, see data structure for specifics.

Each biquad has the form:

$$Y = X(0)*b0 + (b1 * X(-1)) + (b2 * X(-2)) - (a1 * Y(-1)) - (a2 * Y(-2))$$

Preconditions

Delay register values should be initialized to zero. pFilter_Array must contain N*M pFilter elements

Parameters

Parameters	Description
pFilter_Array	pointer to filter coef and delay array structure
Xin	input data element X (int16_t)
N	Number of parallel IIR filter elements (int)
Scale	Scaling factor on each filter element (int32_t)
M	Number of cascaded series BQ filters per parallel element (int)

Returns

Sample output Y ([libq_q15_t](#))

Remarks

Requires N*M filter structures initialized with coefficients pointed to in an array, pFilterArray[0], pFilterArray[1] ... pFilterArray[M] apply to the first parallel filter segment. [M+1], [M+2] ... [M*2] apply to the second segment and so forth.

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass.

An Alpha value of 2 (log2Alpha=1) has been hard coded into the function. This implies that all coefficients should be input at half value. This guarantees that all coefficients can be represented as Q31 fixed point. -1

Example

```
#define NBANDS 9 // number of parallel elements
#define MFILTERS 4
libq_q31_t scaleValue = (1.0/NBANDS)*(1<<31); // Q0.31 scale value = 1/#bands

libq_q15_t Xin,Yout;
EQUALIZER_FILTER myFilterArray[ ] = { BAND1_FILTER1, BAND1_FILTER2, BAND1_FILTER3,
BAND1_FILTER4,
BAND2_FILTER1, BAND2_FILTER2, BAND2_FILTER3,
BAND2_FILTER4,
BAND3_FILTER1, BAND3_FILTER2, BAND3_FILTER3,
BAND3_FILTER4,
BAND4_FILTER1, BAND4_FILTER2, BAND4_FILTER3,
BAND4_FILTER4,
BAND5_FILTER1, BAND5_FILTER2, BAND5_FILTER3,
BAND5_FILTER4,
BAND6_FILTER1, BAND6_FILTER2, BAND6_FILTER3,
BAND6_FILTER4,
BAND7_FILTER1, BAND7_FILTER2, BAND7_FILTER3,
BAND7_FILTER4,
```

```

BAND8_FILTER4,
BAND8_FILTER1, BAND8_FILTER2, BAND8_FILTER3,
BAND9_FILTER4 };
// Normalize filter gains
AUDIO_EQUALIZER_GainNormalizeQ15( NBANDS, MFILTERSterArray );

while ( bGotData )
{
    // Get Xin, filter it
    Yout = AUDIO_EQUALIZER_ParallelNxMinQ15( myFilterArray, Xin, NBANDS, scaleValue, MFILTERS );
    // Play Yout
}

```

File Name

audio_eq_parallelnxm_q15.s

1.9.4.8 AUDIO_EQUALIZER_ParallelNxMinQ31 Function

C

```

libq_q31_t AUDIO_EQUALIZER_ParallelNxMinQ31(
    EQUALIZER_FILTER_32 * pFilter_Array,
    libq_q31_t Xin,
    int N,
    libq_q31_t Scale,
    int M
);

```

Description

Calculates N parallel IIR filters on Xin, sums the result and delivers the result as a 16-bit output. Each parallel filter is a cascade of M biquad IIR filters in series each, for a total of N*M filters calculated. All math is performed using 32 bit instructions, with results truncated to 16-bits for the output. The delay register is stored as a 32-bit value for subsequent functions. Output is tuned by 2 multiplier factors. First each parallel section has a fractional gain (attenuation) that enables individual scaling of that section. Second, a global binary (log2N) gain is applied to the result. The combination of gain factors enable both gain and attenuation.

To normalize the output scale is used as a constant multiply for each parallel block. This value is normally input as 1/N. This is a Q31 numerical value to maintain internal resolution.

All values are fractional Q31, see data structure for specifics.

Each biquad has the form:

$$Y = X(0)*b0 + (b1 * X(-1)) + (b2 * X(-2)) - (a1 * Y(-1)) - (a2 * Y(-2))$$

Preconditions

Delay register values should be initialized to zero. pFilter_Array must contain N*M [EQUALIZER_FILTER_32](#) elements

Parameters

Parameters	Description
pFilter_Array	pointer to filter coef and delay array structure
Xin	input data element X (int16_t)
N	Number of parallel IIR filter elements (int)
Scale	Scaling factor on each filter element (int32_t)
M	Number of cascaded series BQ filters per parallel element (int)

Returns

Sample output Y ([libq_q31_t](#))

Remarks

Requires N*M filter structures initialized with coefficients pointed to in an array, pFilterArray[0], pFilterArray[1] ... pFilterArray[M] apply to the first parallel filter segment. [M+1], [M+2] ... [M*2] apply to the second segment and so forth.

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass.

An Alpha value of 2 (log2Alpha=1) has been hard coded into the function. This implies that all coefficients should be input at half value. This guarantees that all coefficients can be represented as Q31 fixed point. -1

Example

```
#define NBANDS 9          // number of parallel elements
#define MFILTERS 4
libq_q31_t scaleValue = (1.0/NBANDS)*(1<<31); // Q0.31 scale value = 1/#bands

libq_q31_t Xin,Yout;
EQUALIZER_FILTER_32 myFilterArray[ ]={ BAND1_FILTER1, BAND1_FILTER2, BAND1_FILTER3,
BAND1_FILTER4,
                                BAND2_FILTER1, BAND2_FILTER2, BAND2_FILTER3,
BAND2_FILTER4,
                                BAND3_FILTER1, BAND3_FILTER2, BAND3_FILTER3,
BAND3_FILTER4,
                                BAND4_FILTER1, BAND4_FILTER2, BAND4_FILTER3,
BAND4_FILTER4,
                                BAND5_FILTER1, BAND5_FILTER2, BAND5_FILTER3,
BAND5_FILTER4,
                                BAND6_FILTER1, BAND6_FILTER2, BAND6_FILTER3,
BAND6_FILTER4,
                                BAND7_FILTER1, BAND7_FILTER2, BAND7_FILTER3,
BAND7_FILTER4,
                                BAND8_FILTER1, BAND8_FILTER2, BAND8_FILTER3,
BAND8_FILTER4,
                                BAND9_FILTER1, BAND9_FILTER2, BAND9_FILTER3,
BAND9_FILTER4 };
// Normalize filter gains
AUDIO_EQUALIZER_GainNormalizeQ31( NBANDS, MFILTERSterArray );

while ( bGotData )
{
    // Get Xin, filter it
    Yout = AUDIO_EQUALIZER_ParallelNxMinQ31( myFilterArray, Xin, NBANDS, scaleValue, MFILTERS );
    // Play Yout
}
```

File Name

audio_eq_parallelnxm_q31.s

1.9.5 5) Filter Gain Routines

1.9.5.1 AUDIO_EQUALIZER_FilterGainAdjustQ15 Function

C

```
EQUALIZER_FILTER_GAIN AUDIO_EQUALIZER_FilterGainAdjustQ15(
    EQUALIZER_FILTER_GAIN filterGain,
    int16_t gainAdjustment
);
```


Description

Adjusts a filter gain structure by the integer gain adjustment provided

Preconditions

None.

Parameters

Parameters	Description
filterGain	filter gain structure to be adjusted
gainAdjustment	Integer dB adjustment to be applied to the input filter gain

Returns

[EQUALIZER_FILTER_GAIN](#) gain structure after the adjustment is applied

Remarks

For most filtering primitives only the last filter, iFilter = nFiltersPerBand, has a gain that is actually applied. All other gains are ignored.

Example

```
#define NUM_BANDS 6
#define NFILTERS_PER_BAND 2
EQUALIZER_FILTER_GAIN bandGain, adjBandGain;

// Get from user: Band of Interest: iBand, 1<= iBand <= NUM_BANDS
// Get from user: Band Gain Adjustment in dB: iGainAdj, -50 <= iGainAdj <= +50

// Apply gain adjustment to last filter in band cascade
bandGain = AUDIO_EQUALIZER_FilterGainGetQ15(&myFilters[0],
                                             NUM_BANDS, NFILTERS_PER_BAND,
                                             iBand, NFILTERS_PER_BAND );
adjBandGain = AUDIO_EQUALIZER_FilterGainAdjustQ15(bandGain, iGainAdj);
AUDIO_EQUALIZER_FilterGainSetQ15(&myFilters[0],
                                 NUM_BANDS, NFILTERS_PER_BAND,
                                 iBand, NFILTERS_PER_BAND, adjBandGain );
```

1.9.5.2 AUDIO_EQUALIZER_FilterGainAdjustQ31 Function

C

```
EQUALIZER_FILTER_GAIN_32 AUDIO_EQUALIZER_FilterGainAdjustQ31(
    EQUALIZER_FILTER_GAIN_32 filterGain,
    int16_t gainAdjustment
);
```

Description

Adjusts a filter gain structure by the integer gain adjustment provided

Preconditions

None.

Parameters

Parameters	Description
filterGain	filter gain structure to be adjusted
gainAdjustment	Integer dB adjustment to be applied to the input filter gain

Returns

EQUALIZER_FILTER_GAIN_32 gain structure after the adjustment is applied

Remarks

For most filtering primitives only the last filter, `iFilter = nFiltersPerBand`, has a gain that is actually applied. All other gains are ignored.

Example

```
#define NUM_BANDS 6
#define NFILTERS_PER_BAND 2
EQUALIZER_FILTER_GAIN_32 bandGain, adjBandGain;

// Get from user: Band of Interest: iBand, 1<= iBand <= NUM_BANDS
// Get from user: Band Gain Adjustment in dB: iGainAdj, -50 <= iGainAdj <= +50

// Apply gain adjustment to last filter in band cascade
bandGain = AUDIO_EQUALIZER_FilterGainGetQ31(&myFilters[0],
                                             NUM_BANDS, NFILTERS_PER_BAND,
                                             iBand, NFILTERS_PER_BAND );
adjBandGain = AUDIO_EQUALIZER_FilterGainAdjustQ31(bandGain, iGainAdj);
AUDIO_EQUALIZER_FilterGainSetQ31(&myFilters[0],
                                 NUM_BANDS, NFILTERS_PER_BAND,
                                 iBand, NFILTERS_PER_BAND, adjBandGain );
```

1.9.5.3 AUDIO_EQUALIZER_FilterGainGetQ15 Function

C

```
EQUALIZER_FILTER_GAIN AUDIO_EQUALIZER_FilterGainGetQ15(
    EQUALIZER_FILTER * pFilterArray,
    uint16_t nBands,
    uint16_t nFiltersPerBand,
    uint16_t iBand,
    uint16_t iFilter
);
```

Description

Gets the filter gain for a given band and filter.

Preconditions

None.

Parameters

Parameters	Description
<code>pFilterArray</code>	Pointer to first filter in filter array
<code>nBands</code>	number of frequency bands
<code>nFiltersPerBand</code>	number of filters in filter array per each band
<code>iBand</code>	Band of interest, 1,1, ... nBands
<code>iFilter</code>	Filter of interest, 1, ... nFiltersPerBand

Returns

EQUALIZER_FILTER_GAIN gain structure for band/filter of interest

Remarks

For most filtering primitives only the last filter, `iFilter = nFiltersPerBand`, has a gain that is actually applied. All other gains are ignored.

Example

```

#define NUM_BANDS 6
#define NFILTERS_PER_BAND 2
EQUALIZER_FILTER_GAIN bandGain,adjBandGain;

// Get from user: Band of Interest: iBand, 1<= iBand <= NUM_BANDS
// Get from user: Band Gain Adjustment in dB: iGainAdj, -50 <= iGainAdj <= +50

// Apply gain adjustment to last filter in band cascade
bandGain = AUDIO_EQUALIZER_FilterGainGetQ15(&myFilters[0],
                                             NUM_BANDS, NFILTERS_PER_BAND,
                                             iBand, NFILTERS_PER_BAND );
adjBandGain = AUDIO_EQUALIZER_FilterGainAdjustQ15(bandGain,iGainAdj);
AUDIO_EQUALIZER_FilterGainSetQ15(&myFilters[0],
                                 NUM_BANDS, NFILTERS_PER_BAND,
                                 iBand, NFILTERS_PER_BAND, adjBandGain );

```

1.9.5.4 AUDIO_EQUALIZER_FilterGainSetQ15 Function

C

```

void AUDIO_EQUALIZER_FilterGainSetQ15(
    EQUALIZER_FILTER * pFilterArray,
    uint16_t nBands,
    uint16_t nFiltersPerBand,
    uint16_t iBand,
    uint16_t iFilter,
    EQUALIZER_FILTER_GAIN myNewGain
);

```

Description

Gets the filter gain for a given band and filter.

Preconditions

None.

Parameters

Parameters	Description
pFilterArray	Pointer to first filter in filter array
nBands	number of frequency bands
nFiltersPerBand	number of filters in filter array per each band
iBand	Band of interest, 1,1, ... nBands
iFilter	Filter of interest, 1, ... nFiltersPerBand myNewGain EQUALIZER_FILTER_GAIN structure defining new gain values

Returns

None

Remarks

For most filtering primitives only the last filter, iFilter = nFiltlersPerBand, has a gain that is actually applied. All other gains are ignored.

Example

```

#define NUM_BANDS 6
#define NFILTERS_PER_BAND 2
EQUALIZER_FILTER_GAIN bandGain,adjBandGain;

```

```

// Get from user: Band of Interest: iBand, 1<= iBand <= NUM_BANDS
// Get from user: Band Gain Adjustment in dB: iGainAdj, -50 <= iGainAdj <= +50

// Apply gain adjustment to last filter in band cascade
bandGain = AUDIO_EQUALIZER_FilterGainGetQ15(&myFilters[0],
                                             NUM_BANDS, NFILTERS_PER_BAND,
                                             iBand, NFILTERS_PER_BAND );
adjBandGain = AUDIO_EQUALIZER_FilterGainAdjustQ15(bandGain, iGainAdj);
AUDIO_EQUALIZER_FilterGainSetQ15(&myFilters[0],
                                 NUM_BANDS, NFILTERS_PER_BAND,
                                 iBand, NFILTERS_PER_BAND, adjBandGain );

```

1.9.5.5 AUDIO_EQUALIZER_FilterGainGetQ31 Function

C

```

EQUALIZER_FILTER_GAIN_32 AUDIO_EQUALIZER_FilterGainGetQ31(
    EQUALIZER_FILTER_32 * pFilterArray,
    uint16_t nBands,
    uint16_t nFiltersPerBand,
    uint16_t iBand,
    uint16_t iFilter
);

```

Description

Gets the filter gain for a given band and filter.

Preconditions

None.

Parameters

Parameters	Description
pFilterArray	Pointer to first filter in filter array
nBands	number of frequency bands
nFiltersPerBand	number of filters in filter array per each band
iBand	Band of interest, 1,1, ... nBands
iFilter	Filter of interest, 1, ... nFiltersPerBand

Returns

[EQUALIZER_FILTER_GAIN_32](#) gain structure from band/filter of interest

Remarks

For most filtering primitives only the last filter, iFilter = nFiltersPerBand, has a gain that is actually applied. All other gains are ignored.

Example

```

#define NUM_BANDS 6
#define NFILTERS_PER_BAND 2
EQUALIZER_FILTER_GAIN_32 bandGain, adjBandGain;

// Get from user: Band of Interest: iBand, 1<= iBand <= NUM_BANDS
// Get from user: Band Gain Adjustment in dB: iGainAdj, -50 <= iGainAdj <= +50

// Apply gain adjustment to last filter in band cascade
bandGain = AUDIO_EQUALIZER_FilterGainGetQ31(&myFilters[0],
                                             NUM_BANDS, NFILTERS_PER_BAND,
                                             iBand, NFILTERS_PER_BAND );
adjBandGain = AUDIO_EQUALIZER_FilterGainAdjustQ31(bandGain, iGainAdj);
AUDIO_EQUALIZER_FilterGainSetQ31(&myFilters[0],

```

```
NUM_BANDS, NFILTERS_PER_BAND,
iBand, NFILTERS_PER_BAND, adjBandGain );
```

1.9.5.6 AUDIO_EQUALIZER_FilterGainSetQ31 Function

C

```
void AUDIO_EQUALIZER_FilterGainSetQ31(
    EQUALIZER_FILTER_32 * pFilterArray,
    uint16_t nBands,
    uint16_t nFiltersPerBand,
    uint16_t iBand,
    uint16_t iFilter,
    EQUALIZER_FILTER_GAIN_32 myNewGain
);
```

Description

Gets the filter gain for a given band and filter.

Preconditions

None.

Parameters

Parameters	Description
pFilterArray	Pointer to first filter in filter array
nBands	number of frequency bands
nFiltersPerBand	number of filters in filter array per each band
iBand	Band of interest, 1,1, ... nBands
iFilter	Filter of interest, 1, ... nFiltersPerBand myNewGain EQUALIZER_FILTER_GAIN_32 structure defining new gain values

Returns

None

Remarks

For most filtering primitives only the last filter, iFilter = nFiltersPerBand, has a gain that is actually applied. All other gains are ignored.

Example

```
#define NUM_BANDS 6
#define NFILTERS_PER_BAND 2
EQUALIZER_FILTER_GAIN_32 bandGain, adjBandGain;

// Get from user: Band of Interest: iBand, 1<= iBand <= NUM_BANDS
// Get from user: Band Gain Adjustment in dB: iGainAdj, -50 <= iGainAdj <= +50

// Apply gain adjustment to last filter in band cascade
bandGain = AUDIO_EQUALIZER_FilterGainGetQ31(&myFilters[0],
                                           NUM_BANDS, NFILTERS_PER_BAND,
                                           iBand, NFILTERS_PER_BAND );
adjBandGain = AUDIO_EQUALIZER_FilterGainAdjustQ31(bandGain, iGainAdj);
AUDIO_EQUALIZER_FilterGainSetQ31(&myFilters[0],
                                NUM_BANDS, NFILTERS_PER_BAND,
                                iBand, NFILTERS_PER_BAND, adjBandGain );
```

1.9.5.7 AUDIO_EQUALIZER_GainNormalizeQ15 Function

C

```
uint16_t AUDIO_EQUALIZER_GainNormalizeQ15(
    uint16_t nBands,
    uint16_t nFilters,
    EQUALIZER_FILTER * pFilterArray
);
```

Description

Normalize all the [EQUALIZER_FILTER_GAIN](#)'s in a filter array so that the gains can be applied correctly by each filtering function.

For the filters in a frequency band only the gain of the last filter in the band's cascade is applied. Thus the product of all the filters in the cascade is used as the gain of the last filter and all the other filters have a gain set to unity. (Unity gain is gainFrac = 0.5 and gainExp = 1.)

The gains across frequency bands must be adjusted so that there is a common exponent. So the gainFrac's are adjusted to make the gainExp's all the same value. Thus the expGain shift can be postponed to after the calculation of the function's output.

Preconditions

None.

Parameters

Parameters	Description
nBands	number of frequency bands
nFiltersPerBand	number of filters in filter array per each band.
pFilterArray	Pointer to first filter in filter array

Returns

expGain - Normalized gain exponent for all filters in the filter array that has more than one band. For filters that have only one band (nBands = 1) the value returned is zero and can be ignored.

Remarks

The order of the filters when there are multiple filters per band is { Band1_Filter1, Band1_Filter2, ..., Band2_Filter1, Band2_Filter2, ... BandM_Filter1, ..., BandM_FilterN }

Example

A single band with multiple filters:

```
#define FILTER1 {
    {fracGain,expGain}, log2Alpha,
    {b0, b1, b2}, {a1, a2},
    { 0L, 0L } // Z1, Z2 initial values
}

.
.
.
#define FILTER8 {
    {fracGain,expGain}, log2Alpha,
    {b0, b1, b2}, {a1, a2},
    { 0L, 0L } // Z1, Z2 initial values
}

libq_q15_t Xin,Yout;
EQUALIZER_FILTER myFilterArray[ ] = { FILTER1, FILTER2, FILTER3, FILTER4,
```

```

FILTER5, FILTER6, FILTER7, FILTER8 };

// Normalize filter gains, rolling up all gains into last filter.
AUDIO_EQUALIZER_GainNormalizeQ15( 1, 8, myFilterArray );

while ( 1 )
{
    // get new Xin;

    // Filter to get Yout
    Yout = AUDIO_EQUALIZER_Cascade8inQ15( myFilterArray, Xin );
    // Play Yout;
}

```

Multiple bands with 2 IIRs per band:

```

// mBands = 4, nFiltersPerBand = 2
#define BAND1_FILTER1 {
    {fracGain,expGain}, log2Alpha,
    {b0, b1, b2}, {a1, a2},
    { 0L, 0L } // Z1, Z2 initial values
}

.
.
.
#define BAND4_FILTER2 {
    {fracGain,expGain}, log2Alpha,
    {b0, b1, b2}, {a1, a2},
    { 0L, 0L } // Z1, Z2 initial values
}

libq_q15_t Xin,Yout;
EQUALIZER_FILTER myFilterArray[ ] = {  BAND1_FILTER1, BAND1_FILTER2,
                                       BAND2_FILTER1, BAND2_FILTER2,
                                       BAND3_FILTER1, BAND3_FILTER2,
                                       BAND4_FILTER1, BAND4_FILTER2 };

// Normalize filter gain exponent.
AUDIO_EQUALIZER_GainNormalizeQ15( 4, 2, myFilterArray );

while ( 1 )
{
    // get new Xin;

    // Filter to get Yout
    Yout = AUDIO_EQUALIZER_Parallel4x2inQ15(myFilterArray, Xin );
    // Play Yout;
}

```

1.9.5.8 AUDIO_EQUALIZER_GainNormalizeQ31 Function

C

```

uint16_t AUDIO_EQUALIZER_GainNormalizeQ31(
    uint16_t nBands,
    uint16_t nFiltersPerBand,
    EQUALIZER_FILTER_32 * pFilterArray
);

```

Description

Normalize all the **EQUALIZER_FILTER_GAIN**'s in a filter array so that the gains can be applied correctly by each filtering function.

For the filters in a frequency band only the gain of the last filter in the band's cascade is applied. Thus the product of all the filters in the cascade is used as the gain of the last filter and all the other filters have a gain set to unity. (Unity gain is gainFrac = 0.5

and gainExp = 1.)

The gains across frequency bands must be adjusted so that there is a common exponent. So the gainFrac's are adjusted to make the gainExp's all the same value. Thus the expGain shift can be postponed to after the calculation of the function's output.

Preconditions

None.

Parameters

Parameters	Description
nBands	number of frequency bands
nFiltersPerBand	number of filters in filter array per each band.
pFilterArray	Pointer to first filter in filter array

Returns

expGain - Normalized gain exponent for all filters in the filter array that has more than one band. For filters that have only one band (nBands = 1) the value returned is zero.

Remarks

The order of the filters when there are multiple filters per band is { Band1_Filter1, Band1_Filter2, ..., Band2_Filter1, Band2_Filter2, ... BandM_Filter1, ..., BandM_FilterN }

Example

A single band with multiple filters:

```
#define FILTER1 {
    {fracGain,expGain}, log2Alpha,
    {b0, b1, b2}, {a1, a2},
    { 0L, 0L } // Z1, Z2 initial values
}

.
.
.
#define FILTER8 {
    {fracGain,expGain}, log2Alpha,
    {b0, b1, b2}, {a1, a2},
    { 0L, 0L } // Z1, Z2 initial values
}

libq_q31_t Xin,Yout;
EQUALIZER_FILTER myFilterArray[ ] = { FILTER1, FILTER2, FILTER3, FILTER4,
                                       FILTER5, FILTER6, FILTER7, FILTER8 };

// Normalize filter gains, rolling up all gains into last filter.
AUDIO_EQUALIZER_GainNormalizeQ31( 1, 8, myFilterArray );

while ( 1 )
{
    // get new Xin;

    // Filter to get Yout
    Yout = AUDIO_EQUALIZER_Cascade8inQ31( myFilterArray, Xin );
    // Play Yout;
}
```

Multiple bands with 2 IIRs per band:

```
// mBands = 4, nFiltersPerBand = 2
#define BAND1_FILTER1 {
    {fracGain,expGain}, log2Alpha,
    {b0, b1, b2}, {a1, a2},
```



```

        { 0L, 0L } // Z1, Z2 initial values
    }
    .
    .
    .
#define BAND4_FILTER2 {
    {fracGain,expGain}, log2Alpha,
    {b0, b1, b2}, {a1, a2},
    { 0L, 0L } // Z1, Z2 initial values
}

libq_q31_t Xin,Yout;
EQUALIZER_FILTER myFilterArray[ ] = {  BAND1_FILTER1, BAND1_FILTER2,
                                       BAND2_FILTER1, BAND2_FILTER2,
                                       BAND3_FILTER1, BAND3_FILTER2,
                                       BAND4_FILTER1, BAND4_FILTER2 };

// Normalize filter gain exponent.
AUDIO_EQUALIZER_GainNormalizeQ31( 4, 2, myFilterArray );

while ( 1 )
{
    // get new Xin;

    // Filter to get Yout
    Yout = AUDIO_EQUALIZER_Parallel4x2inQ31(myFilterArray, Xin );
    // Play Yout;
}

```

1.9.6 6) Band Energy Estimation

1.9.6.1 AUDIO_EQUALIZER_BandEnergySumsInit Function

C

```

void AUDIO_EQUALIZER_BandEnergySumsInit(
    uint16_t nBands,
    BAND_ENERGY_UNITS units
);

```

Description

Initialize band energy measurements, clearing band energy sum array and number of energy samples for each band. Second argument determines whether signal energy is measured using RMS or pseudo RMS.

Preconditions

None.

Parameters

Parameters	Description
nBands	number of filter bands in use, must be <= AUDIO_EQUALIZER_MAX_NBANDS
units	BAND_ENERGY_RMS_VOLTS, BAND_ENERGY_RMS_DBFS, BAND_ENERGY_PSEUDORMS_VOLTS, or BAND_ENERGY_PSEUDORMS_DBFS

Returns

None.

Remarks

None.

Example

```
AUDIO_EQUALIZER_BandEnergySumsInit( 6, BAND_ENERGY_RMS_DBFS );
```

1.9.6.2 AUDIO_EQUALIZER_BandEnergyNSamplesSet Function

C

```
void AUDIO_EQUALIZER_BandEnergyNSamplesSet (
    uint16_t nSamples
);
```

Description

Resets number of samples used to update band energy measurements. This routine is not needed if [AUDIO_EQUALIZER_BandEnergyUpdateQ15/Q31](#) is used to update band energy estimates with new samples. It is used when band energy sums are updated directly via access to the extern `AUDIO_EQUALIZER_BandEnergySumQ15` array.

Preconditions

[AUDIO_EQUALIZER_BandEnergySumsInit](#) has been called to setup the number of bands and the units of band energy.

Parameters

Parameters	Description
nSamples	value of number of samples. Use zero to reinitialize the number of samples.

Returns

None.

Remarks

None.

Example

Band energy measurements are initialized by:

```
#define NUM_BANDS 6
AUDIO_EQUALIZER_BandEnergySumsInit( 2*NUM_BANDS, BAND_ENERGY_PSEUDORMS_DBFS );
```

Here is an example filtering primitive that directly updates band energy sums to reduce its cycle count. The `AUDIO_EQUALIZER_nBandSamples` array is not updated to save cycles. Instead `AUDIO_EQUALIZER_nSamples` is called.

```
#include <stdint.h>
#include "math/audio_equalizer/audio_equalizer.h"

extern uint16_t AUDIO_EQUALIZER_nSamples;
extern libq_q16d15_t AUDIO_EQUALIZER_BandEnergySumQ15[AUDIO_EQUALIZER_MAX_NBANDS];

#include "myFilters6x2_Q15_Stereo.h"

void FilterInput(libq_q15_t XinLeft, libq_q15_t XinRight, libq_q15_t *YoutLeft,
libq_q15_t *YoutRight)
{
    libq_q15_t Yout0,Yout1,Yout2,Yout3,Yout4,Yout5;

    Yout0 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[0], XinLeft );
    AUDIO_EQUALIZER_BandEnergySumQ15[0] += abs(Yout0);

    Yout1 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[2], XinLeft );
```

```

    AUDIO_EQUALIZER_BandEnergySumQ15[1] += abs(Yout1);

    Yout2 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[4], XinLeft );
    AUDIO_EQUALIZER_BandEnergySumQ15[2] += abs(Yout2);

    Yout3 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[6], XinLeft );
    AUDIO_EQUALIZER_BandEnergySumQ15[3] += abs(Yout3);

    Yout4 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[8], XinLeft );
    AUDIO_EQUALIZER_BandEnergySumQ15[4] += abs(Yout4);

    Yout5 = AUDIO_EQUALIZER_Cascade2inQ15(&myFiltersLeft[10], XinLeft );
    AUDIO_EQUALIZER_BandEnergySumQ15[5] += abs(Yout5);

    *YoutLeft = Yout0 + Yout1 + Yout2 + Yout3 + Yout4 + Yout5;

    Yout0 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[0], XinRight );
    AUDIO_EQUALIZER_BandEnergySumQ15[6] += abs(Yout0);

    Yout1 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[2], XinRight );
    AUDIO_EQUALIZER_BandEnergySumQ15[7] += abs(Yout1);

    Yout2 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[4], XinRight );
    AUDIO_EQUALIZER_BandEnergySumQ15[8] += abs(Yout2);

    Yout3 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[6], XinRight );
    AUDIO_EQUALIZER_BandEnergySumQ15[9] += abs(Yout3);

    Yout4 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[8], XinRight );
    AUDIO_EQUALIZER_BandEnergySumQ15[10] += abs(Yout4);

    Yout5 = AUDIO_EQUALIZER_Cascade2inQ15(&myFiltersRight[10], XinRight );
    AUDIO_EQUALIZER_BandEnergySumQ15[11] += abs(Yout5);

    *YoutRight = Yout0 + Yout1 + Yout2 + Yout3 + Yout4 + Yout5;

    AUDIO_EQUALIZER_nSamples += 1;
}

```

Band energy measurements are retrieved using [AUDIO_EQUALIZER_BandEnergyGetQ15](#) or [AUDIO_EQUALIZER_BandEnergyGetQ31](#). After retrieving all the measurements, `AUDIO_EQUALIZER_BandEnergyNSamplesSet(0)` is called to reinitialize the sample count.

```

libq_q15d16_t bandEnergyInDB[NUM_BANDS];
uint16_t iBand;
for ( iBand = 0; iBand < NUM_BANDS; iBand++ )
{
    // Get band energy measurement, clear band energy sum
    bandEnergyInDB[iBand] = AUDIO_EQUALIZER_BandEnergyGetQ15(iBand, true);
}
AUDIO_EQUALIZER_BandEnergyNSamplesSet(0); // Clear sample counter

```

1.9.6.3 AUDIO_EQUALIZER_BandEnergyUpdateQ15 Function

C

```

void AUDIO_EQUALIZER_BandEnergyUpdateQ15(
    uint16_t iBand,
    libq_q15_t YoutQ15
);

```

Description

Update band energy estimate for a given filter band with new filter output. Energy sum array element corresponding to `iBand` is updated with `Yout^2` or `abs(Yout)` depending on value of units used when energy sum buffer was initialized to zero using [AUDIO_EQUALIZER_BandEnergySumsInit](#).

"Q15" suffix designates this routine is for signals with Q15 fixed point format.

Preconditions

[AUDIO_EQUALIZER_BandEnergySumsInit](#) has been called.

Parameters

Parameters	Description
iBand	band index, from 0 to nBands - 1, where nBands was the value used in call to AUDIO_EQUALIZER_BandEnergySumsInit
YoutQ15	Q15 filter output for band iBand

Returns

none

Remarks

None.

Example

```
#include <stdint.h>
#include "math/audio_equalizer/audio_equalizer.h"

#include "myFilters6x2_Q15_Stereo.h"

void FilterInput(libq_q15_t XinLeft, libq_q15_t XinRight, libq_q15_t *YoutLeft, libq_q15_t
*YoutRight)
{
    libq_q15_t Yout0,Yout1,Yout2,Yout3,Yout4,Yout5;

    Yout0 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[0], XinLeft );
    AUDIO_EQUALIZER_BandEnergyUpdateQ15(0,Yout0);

    Yout1 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[2], XinLeft );
    AUDIO_EQUALIZER_BandEnergyUpdateQ15(1,Yout1);

    Yout2 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[4], XinLeft );
    AUDIO_EQUALIZER_BandEnergyUpdateQ15(2,Yout2);

    Yout3 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[6], XinLeft );
    AUDIO_EQUALIZER_BandEnergyUpdateQ15(3,Yout3);

    Yout4 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersLeft[8], XinLeft );
    AUDIO_EQUALIZER_BandEnergyUpdateQ15(4,Yout4);

    Yout5 = AUDIO_EQUALIZER_Cascade2inQ15(&myFiltersLeft[10], XinLeft );
    AUDIO_EQUALIZER_BandEnergyUpdateQ15(5,Yout5);

    *YoutLeft = Yout0 + Yout1 + Yout2 + Yout3 + Yout4 + Yout5;

    Yout0 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[0], XinRight );
    AUDIO_EQUALIZER_BandEnergyUpdateQ15(6,Yout0);

    Yout1 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[2], XinRight );
    AUDIO_EQUALIZER_BandEnergyUpdateQ15(7,Yout1);

    Yout2 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[4], XinRight );
    AUDIO_EQUALIZER_BandEnergyUpdateQ15(8,Yout2);

    Yout3 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[6], XinRight );
    AUDIO_EQUALIZER_BandEnergyUpdateQ15(9,Yout3);

    Yout4 = AUDIO_EQUALIZER_Cascade2inQ15( &myFiltersRight[8], XinRight );
    AUDIO_EQUALIZER_BandEnergyUpdateQ15(10,Yout4);
```

```

    Yout5 = AUDIO_EQUALIZER_Cascade2inQ15(&myFiltersRight[10], XinRight );
    AUDIO_EQUALIZER_BandEnergyUpdateQ15(11,Yout5);

    *YoutRight = Yout0 + Yout1 + Yout2 + Yout3 + Yout4 + Yout5;
}

```

1.9.6.4 AUDIO_EQUALIZER_BandEnergyUpdateQ31 Function

C

```

void AUDIO_EQUALIZER_BandEnergyUpdateQ31 (
    uint16_t iBand,
    libq_q31_t YoutQ31
);

```

Description

Update band energy estimate for a given filter band with new filter output. Energy sum array element corresponding to iBand is updated with Yout^2 or abs(Yout) depending on value of units used when energy sum buffer was initialized to zero using [AUDIO_EQUALIZER_BandEnergySumsInit](#).

"Q31" suffix designates this routine is for signals with Q31 fixed point format.

Preconditions

[AUDIO_EQUALIZER_BandEnergySumsInit](#) has been called.

Parameters

Parameters	Description
iBand	band index, from 0 to nBands - 1, where nBands was the value used in call to AUDIO_EQUALIZER_BandEnergySumsInit
YoutQ31	Q31 filter output for band iBand

Returns

none

Remarks

None.

Example

```

void FilterInput(libq_q31_t XinLeft, libq_q31_t XinRight, libq_q31_t *YoutLeft, libq_q31_t
*YoutRight)
{
    libq_q31_t Yout0,Yout1,Yout2,Yout3,Yout4,Yout5;

    Yout0 = AUDIO_EQUALIZER_Cascade2inQ31( &myFiltersLeft[0], XinLeft );
    AUDIO_EQUALIZER_BandEnergyUpdateQ31(0,Yout0);

    Yout1 = AUDIO_EQUALIZER_Cascade2inQ31( &myFiltersLeft[2], XinLeft );
    AUDIO_EQUALIZER_BandEnergyUpdateQ31(1,Yout1);

    Yout2 = AUDIO_EQUALIZER_Cascade2inQ31( &myFiltersLeft[4], XinLeft );
    AUDIO_EQUALIZER_BandEnergyUpdateQ31(2,Yout2);

    Yout3 = AUDIO_EQUALIZER_Cascade2inQ31( &myFiltersLeft[6], XinLeft );
    AUDIO_EQUALIZER_BandEnergyUpdateQ31(3,Yout3);

    Yout4 = AUDIO_EQUALIZER_Cascade2inQ31( &myFiltersLeft[8], XinLeft );
    AUDIO_EQUALIZER_BandEnergyUpdateQ31(4,Yout4);

    Yout5 = AUDIO_EQUALIZER_Cascade2inQ31(&myFiltersLeft[10], XinLeft );
}

```

```

AUDIO_EQUALIZER_BandEnergyUpdateQ31(5,Yout5);

*YoutLeft = Yout0 + Yout1 + Yout2 + Yout3 + Yout4 + Yout5;

Yout0 = AUDIO_EQUALIZER_Cascade2inQ31( &myFiltersRight[0], XinRight );
AUDIO_EQUALIZER_BandEnergyUpdateQ31(6,Yout0);

Yout1 = AUDIO_EQUALIZER_Cascade2inQ31( &myFiltersRight[2], XinRight );
AUDIO_EQUALIZER_BandEnergyUpdateQ31(7,Yout1);

Yout2 = AUDIO_EQUALIZER_Cascade2inQ31( &myFiltersRight[4], XinRight );
AUDIO_EQUALIZER_BandEnergyUpdateQ31(8,Yout2);

Yout3 = AUDIO_EQUALIZER_Cascade2inQ31( &myFiltersRight[6], XinRight );
AUDIO_EQUALIZER_BandEnergyUpdateQ31(9,Yout3);

Yout4 = AUDIO_EQUALIZER_Cascade2inQ31( &myFiltersRight[8], XinRight );
AUDIO_EQUALIZER_BandEnergyUpdateQ31(10,Yout4);

Yout5 = AUDIO_EQUALIZER_Cascade2inQ31(&myFiltersRight[10], XinRight );
AUDIO_EQUALIZER_BandEnergyUpdateQ31(11,Yout5);

*YoutRight = Yout0 + Yout1 + Yout2 + Yout3 + Yout4 + Yout5;
}

```

1.9.6.5 AUDIO_EQUALIZER_BandEnergyGetQ15 Function

C

```

libq_q15d16_t AUDIO_EQUALIZER_BandEnergyGetQ15(
    uint16_t iBand,
    bool bEnergySumClear
);

```

Description

Get band energy estimate for a given filter band. Estimate is either RMS or pseudo RMS, depending on the units value used when [AUDIO_EQUALIZER_BandEnergySumsInit](#) was called to initialize/reinitialize the energy sum array.

"Q15" suffix designates this routine is for signals with Q15 fixed point format.

Preconditions

[AUDIO_EQUALIZER_BandEnergySumsInit](#) has been called.

Parameters

Parameters	Description
iBand	band index, from 0 to nBands - 1, where nBands was the value used in call to AUDIO_EQUALIZER_BandEnergySumsInit
bEnergySumClear	true -> clear corresponding energy sum array element to zero

Returns

Signal energy estimate for a given filter band as Q15.16. The meaning of the 32 bit integer depends on the units chosen when [AUDIO_EQUALIZER_BandEnergySumsInit](#) was first called. See the Remarks section for more information.

Remarks

Q15.16 format is a 32-bit word, consisting of 16 fractional bits (least significant word) with the sign bit (MSB) and 15 bits in the most significant word to store the integer portion.

If X is Q15.16 format then $-2^{15} \leq X \leq 2^{15} - 2^{-16}$, or simply $-32678 \leq X \leq 32768 - 1/65536$

The meaning of the returned value depends on the units chosen when [AUDIO_EQUALIZER_BandEnergySumsInit](#) was first

called. See the code example above for the details.

Example

If units==BAND_ENERGY_RMS_DBFS or units==BAND_ENERGY_PSEUDORMS_DBFS then

```
libq_q15d16_t bandEnergyInDB = AUDIO_EQUALIZER_BandEnergyGetQ15(1,true);
libq_q15d16_t absBandEnergyInDB;
int16_t intBandEnergyInDB;
libq_q0d16_t fracBandEnergyInDB;
float floatBandEnergyInDB;

if ( bandEnergyInDB > 0 )
{
    // Error, band energy in dBFS is always negative or zero!
}
else
{
    floatBandEnergyInDB = bandEnergyInDB/65536.0;
    intBandEnergyInDB = bandEnergyInDB>>16;
    absBandEnergyInDB = labs(bandEnergyInDB);
    fracBandEnergyInDB = (libq_q0d16_t)absBandEnergyInDB; // get lower 16 bits

    // alternative calculation for band energy as floating point number
    floatBandEnergyInDB = intBandEnergyInDB - fracBandEnergyInDB/65536.0;
}
```

If units == BAND_ENERGY_RMS_VOLTS or units == BAND_ENERGY_PSEUDORMS_VOLTS then only the fractional part of a band energy value is significant, the integer part is always zero. Band energy in RMS volts is represented by a 16 bit Q0d16 fixed point number, where 1 represents a signal with all samples at the maximum possible ADC count.

```
libq_q15d16_t bandEnergyInVolts = AUDIO_EQUALIZER_BandEnergyGetQ15(1,true);
libq_q0d16_t bandEnergyQ16;
libq_q0d15_t bandEnergyQ15;
float floatBandEnergyInVolts;

if ( bandEnergyInVolts < 0 )
{
    // Error band energy in volts is never negative!
}
else
{
    // Strip off fractional part of Q15.16, convert to floating point
    bandEnergyQ16 = (libq_q0d16_t)( bandEnergyInVolts & 0x0000FFFF );
    floatBandEnergyInVolts = bandEnergyQ16 / 65536.0;

    //OR - Calculate Q15 band energy and convert to floating point
    bandEnergyQ15 = bandEnergyQ16>>1;
    floatBandEnergyInVolts = bandEnergyQ15 / 32768.0;
}
```

1.9.6.6 AUDIO_EQUALIZER_BandEnergyGetQ31 Function

C

```
libq_q15d16_t AUDIO_EQUALIZER_BandEnergyGetQ31(
    uint16_t iBand,
    bool bEnergySumClear
);
```

Description

Get band energy estimate for a given filter band. Estimate is either RMS or pseudo RMS, depending on the units value used when [AUDIO_EQUALIZER_BandEnergySumsInit](#) was called to initialize/reinitialize the energy sum array.

"Q31" suffix designates this routine is for signals with Q31 fixed point format.

Preconditions

[AUDIO_EQUALIZER_BandEnergySumsInit](#) has been called.

Parameters

Parameters	Description
iBand	band index, from 0 to nBands - 1, where nBands was the value used in call to AUDIO_EQUALIZER_BandEnergySumsInit
bEnergySumClear	true -> clear corresponding energy sum array element to zero

Returns

Signal energy estimate for a given filter band as Q15.16. The meaning of the 32 bit integer depends on the units chosen when [AUDIO_EQUALIZER_BandEnergySumsInit](#) was first called.

See the Remarks section for more information.

Remarks

Q15.16 format is a 32-bit word, consisting of 16 fractional bits (least significant word) with the sign bit (MSB) and 15 bits in the most significant word to store the integer portion.

If X is Q15.16 format then $-2^{15} \leq X \leq 2^{15} - 2^{-16}$, or simply $-32768 \leq X \leq 32768 - 1/65536$

The meaning of the returned value depends on the units chosen when [AUDIO_EQUALIZER_BandEnergySumsInit](#) was first called. See the code example above for the details.

Example

If units==BAND_ENERGY_RMS_DBFS or units==BAND_ENERGY_PSEUDORMS_DBFS then

```
libq_q15d16_t bandEnergyInDB = AUDIO_EQUALIZER_BandEnergyGetQ31(1, true);
libq_q15d16_t absBandEnergyInDB;
int16_t      intBandEnergyInDB;
libq_q0d16_t fracBandEnergyInDB;
float        floatBandEnergyInDB;

if ( bandEnergyInDB > 0 )
{
    // Error, band energy in dBFS is always negative or zero!
}
else
{
    floatBandEnergyInDB = bandEnergyInDB/65536.0;
    intBandEnergyInDB = bandEnergyInDB>>16;
    absBandEnergyInDB = labs(bandEnergyInDB);
    fracBandEnergyInDB = (libq_q0d16_t)absBandEnergyInDB; // get lower 16 bits

    // alternative calculation for band energy as floating point number
    floatBandEnergyInDB = intBandEnergyInDB - fracBandEnergyInDB/65536.0;
}
```

If units == BAND_ENERGY_RMS_VOLTS or units == BAND_ENERGY_PSEUDORMS_VOLTS then only the fractional part of a band energy value is significant, the integer part is always zero. Band energy in RMS volts is represented by a 16 bit Q15 fixed point number, where 1 represents a signal with all samples at the maximum possible ADC count.

```
libq_q15d16_t bandEnergyInVolts = AUDIO_EQUALIZER_BandEnergyGetQ31(1, true);
libq_q0d16_t  bandEnergyQ16;
libq_q0d15_t  bandEnergyQ15;
float         floatBandEnergyInVolts;

if ( bandEnergyInVolts < 0 )
{
    // Error band energy in volts is never negative!
```



```

}
else
{
    // Strip off fractional part of Q15.16, convert to floating point
    bandEnergyQ16 = (libq_q0d16_t)( bandEnergyInVolts & 0x0000FFFF );
    floatBandEnergyInVolts = bandEnergyQ16 / 65536;

    //OR - Calculate Q15 band energy and convert to floating point
    bandEnergyQ15 = bandEnergyQ16>>1;
    floatBandEnergyInVolts = bandEnergyQ15 / 32768;
}

```

1.9.7 7) Fixed Point Typedefs

1.9.7.1 libq_q0d15_t Type

C

```
typedef int16_t libq_q0d15_t;
```

Description

Fixed Point Integer Typedef

Typedef for the Q0.15 fixed point data type into a 16 bit signed integer. Values for this data type are in the range [-1,+1), i.e. $-1 \leq x < +1$.

Remarks

This data type has one sign bit and 15 fractional bits in a 16 bit word.

```

-----1-----
5432109876543210
-----
Sfffffffffffffffff

```

Example

```

float Xfloat;
libq_q0d15_t Xq0d15;

Xfloat = Xq0d15/32768.0;

```

1.9.7.2 libq_q15_t Type

C

```
typedef libq_q0d15_t libq_q15_t;
```

Description

Fixed Point Integer Typedef

Typedef for the Q0.15 fixed point data type into a 16 bit signed integer. Values for this data type are in the range [-1,+1), i.e. $-1 \leq x < +1$.

Remarks

This data type has one sign bit and 15 fractional bits in a 16 bit word.

```

-----1-----
5432109876543210
-----
Sfffffffffffffffff

```

Example

```

float Xfloat;
libq_q15_t Xq15;

Xfloat = Xq15/32768.0;

```

1.9.7.3 libq_q0d16_t Type

C

```
typedef uint16_t libq_q0d16_t;
```

Description

Fixed Point Integer Typedef

Typedef for the Q0.16 fixed point data type into a 16 bit unsigned integer. Values for this data type are in the range [0,+1], i.e. $-1 \leq x \leq +1$.

Remarks

This data type has no sign bit and 16 fractional bits in a 16 bit word.

```

-----1-----
5432109876543210
-----
fffffffffffffffff

```

Example

```

float Xfloat;
libq_q0d16_t Xq0d16;

Xfloat = Xq0d16/65536.0;

```

1.9.7.4 libq_q0d31_t Type

C

```
typedef int32_t libq_q0d31_t;
```

Description

Fixed Point Integer Typedef

Typedef for the Q0.15 fixed point data type into a 32 bit signed integer. Values for this data type are in the range $[-1,+1)$, i.e. $-1 \leq x < +1$.

Remarks

This data type has one sign bit and 31 fractional bits in a 32 bit word.

```

-3-----2-----1-----
10987654321098765432109876543210
-----
Sfffffffffffffffffffffffffffffffff

```

Example

```
float Xfloat;
libq_q0d31_t Xq0d31;

Xfloat = Xq0d31/((float)2<<31);
```

1.9.7.5 libq_q31_t Type

C

```
typedef libq_q0d31_t libq_q31_t;
```

Description

Fixed Point Integer Typedef

Typedef for the Q0.15 fixed point data type into a 32 bit signed integer. Values for this data type are in the range [-1,+1), i.e. $-1 \leq x < +1$.

Remarks

This data type has one sign bit and 31 fractional bits in a 32 bit word.

```
-3-----2-----1-----
10987654321098765432109876543210
-----
Sffffffffffffffffffffffffffffffff
```

Example

```
float Xfloat;
libq_q31_t Xq31;

Xfloat = Xq31/((float)2<<31);
```

1.9.7.6 libq_q0d63_t Type

C

```
typedef int64_t libq_q0d63_t;
```

Description

Fixed Point Integer Typedef

Typedef for the Q0.63 fixed point data type into a 64 bit signed integer. Values for this data type are in the range [-1,+1), i.e. $-1 \leq x < +1$.

Remarks

This data type has one sign bit and 63 fractional bits.

```
---6-----5-----4-----3-----2-----1-----
321098765432109876543210987654321098765432109876543210
Sffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

1.9.7.7 libq_q63_t Type

C

```
typedef libq_q0d63_t libq_q63_t;
```

Description

Fixed Point Integer Typedef

Typedef for the Q0.63 fixed point data type into a 64 bit signed integer. Values for this data type are in the range $[-1, +1)$, i.e. $-1 \leq x < +1$.

Remarks

This data type has one sign bit and 63 fractional bits.

```

---6-----5-----4-----3-----2-----1-----
321098765432109876543210987654321098765432109876543210
Sffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

```

1.9.7.8 libq_q15d16_t Type

C

```
typedef int32_t libq_q15d16_t;
```

Description

Fixed Point Integer Typedef

Typedef for the Q15.16 fixed point data type into 32 bit signed integer Values for this data type are in the range $[-32768, +32768)$, i.e. $-32768 \leq x < +32768$.

Remarks

This data type has one sign bit, 15 integer bits, and 16 fractional bits.

```

-3-----2-----1-----
10987654321098765432109876543210
-----
Siiiiiiiiiiiiiiiiifffffffffffffffffffff

```

Example

```

float Xfloat;
libq_q15d16_t Xq15d31;

Xfloat = Xq15d31/65536.0;

```

1.9.7.9 libq_q16d15_t Type

C

```
typedef int32_t libq_q16d15_t;
```

Description

Fixed Point Integer Typedef

Typedef for the Q16d15 fixed point data type into a 32 bit signed integer Values for this data type are in the range $[-65536, +65536)$, i.e. $-65536 \leq x < +65536$.

Remarks

This data type has one sign bit, 16 integer bits, and 15 fractional bits.

```

-3-----2-----1-----
10987654321098765432109876543210
-----
Siiiiiiiiiiiiiiiiifffffffffffffffffffff

```

Example

```
float Xfloat;
libq_q15d16_t Xq15d31;

Xfloat = Xq15d31/32768.0;
```

1.9.8 8) Data Types and Constants

1.9.8.1 AUDIO_EQUALIZER_MAX_NBANDS Macro

C

```
#define AUDIO_EQUALIZER_MAX_NBANDS 16
```

Description

Maximum number of filter bands supported

Maximum number of filter bands supported, used in calculating band signal RMS or Pseudo RMS values. Since this filtering [library](#) does not explicitly support left/right stereo, filter bands must be explicitly allocated. A value of 16 will support up to 8 frequency bands for stereo processing.

Remarks

None.

1.9.8.2 BAND_ENERGY_UNITS Enumeration

C

```
typedef enum {
    BAND_ENERGY_VOLTS_SQUARED,
    BAND_ENERGY_RMS_VOLTS,
    BAND_ENERGY_RMS_DBFS,
    BAND_ENERGY_PSEUDORMS_VOLTS,
    BAND_ENERGY_PSEUDORMS_DBFS
} BAND_ENERGY_UNITS;
```

Description

Band Energy Units Enumeration

Determines what units are used in reporting band energy.

Members

Members	Description
BAND_ENERGY_VOLTS_SQUARED	Band energy reported in RMS Voltage, with Q15 format
BAND_ENERGY_RMS_VOLTS	Band energy reported in RMS Voltage, with Q15 format
BAND_ENERGY_RMS_DBFS	Band energy reported in dB re Full Scale using RMS energy estimate
BAND_ENERGY_PSEUDORMS_VOLTS	Band energy reported in Pseudo RMS Voltage, with Q15 format
BAND_ENERGY_PSEUDORMS_DBFS	Band energy reported in dB re Full Scale using Pseudo RMS energy estimate

Remarks

None.

1.9.8.3 EQUALIZER_FILTER Structure

C

```
typedef struct {
    EQUALIZER_FILTER_GAIN G;
    uint16_t log2Alpha;
    libq_q15_t b[3];
    libq_q15_t a[2];
    int32_t Z[2];
} EQUALIZER_FILTER;
```

Description

Equalizer IIR Filter Definition for Q15 Filtering

Typedef for equalizer IIR filter definition structure. Defines filter taps and gain multiplier.

Members

Members	Description
EQUALIZER_FILTER_GAIN G;	Filter max gain multiplier
uint16_t log2Alpha;	Coefficient scaling bit shift value, for most filters should be at least one.
libq_q15_t b[3];	Feedforward Coefficients, Q15 format
libq_q15_t a[2];	Feedback Coefficients, Q15 format, Always: a0 = 1
int32_t Z[2];	Filter memory, should be initialized to zero.

Remarks

Only C prototypes use log2Alpha. Routines optimized in assembly ignore log2Alpha, assuming it is always one.

Filter coefficients are normalized by dividing by Alpha or equivalently, by a right shift by log2Alpha bits:

```
#define ALPHA      2
#define LOG2ALPHA 1
libq_q15_t a[] = { a1/ALPHA, a2/ALPHA }; // a0 assumed to be 1
libq_q15_t b[] = { b0>>LOG2ALPHA, b1>>LOG2ALPHA, b2>>LOG2ALPHA };
```

1.9.8.4 EQUALIZER_FILTER_32 Structure

C

```
typedef struct {
    EQUALIZER_FILTER_GAIN_32 G;
    uint32_t log2Alpha;
    libq_q31_t b[3];
    libq_q31_t a[2];
    int32_t Z[2];
} EQUALIZER_FILTER_32;
```

Description

Equalizer IIR Filter Definition for Q31 Filtering

Typedef for equalizer IIR filter definition structure. Defines filter taps and gain multiplier.

Members

Members	Description
EQUALIZER_FILTER_GAIN_32 G;	Filter max gain multiplier

uint32_t log2Alpha;	Coefficient scaling bit shift value, (32 bits for sake of alignment in assembly) for most filters should be at least one.
libq_q31_t b[3];	Feedforward Coefficients, Q31 format
libq_q31_t a[2];	Feedback Coefficients, Q31 format, Always: a0 = 1
int32_t Z[2];	Filter memory, should be initialized to zero.

Remarks

Only C prototypes use log2Alpha. Routines optimized in assembly ignore log2Alpha, assuming it is always one.

Filter coefficients are normalized by dividing by Alpha or equivalently, by a right shift by log2Alpha bits:

```
#define ALPHA      2
#define LOG2ALPHA 1
libq_q31_t a[] = { a1/ALPHA, a2/ALPHA }; // a0 assumed to be 1
libq_q31_t b[] = { b0>>LOG2ALPHA, b1>>LOG2ALPHA, b2>>LOG2ALPHA };
```

1.9.8.5 EQUALIZER_FILTER_GAIN Structure

C

```
typedef struct {
    libq_q15_t fracGain;
    uint16_t expGain;
} EQUALIZER_FILTER_GAIN;
```

Description

Equalizer IIR Filter Gain Structure, Q15

Typedef for filter gain structure. Defines filter gain multiplier as block floating point number (mantissa and exponent). Mantissa is Q0.15 (Q15) as an int16_t and the exponent is an unsigned integer (uint16_t).

Remarks

Typical use:

```
EQUALIZER_FILTER_GAIN G = {0x8503,1}; // Filter gain
libq_q15_t Yin,Yout;
libq_q31_t Y32;
Y32 = ((G.fracGain*Yin)<<1)<<G.expGain;
Yout = Y32>>16;
```

In the code above:

G.fracGain*Y16 as fixed point = (G.fracGain*Y16)<<1 as integers.

So <<1 is necessary since all multiplies in C are integer not fixed point.

1.9.8.6 EQUALIZER_FILTER_GAIN_32 Structure

C

```
typedef struct {
    libq_q31_t fracGain;
    uint32_t expGain;
} EQUALIZER_FILTER_GAIN_32;
```

Description

Equalizer IIR Filter Gain Structure, Q32

Typedef for filter gain structure. Defines filter gain multiplier as block floating point number (mantissa and exponent). Mantissa is Q0.15 (Q15) as an int16_t and the exponent is an unsigned integer (uint16_t).

Remarks

Typical use:

```
EQUALIZER_FILTER_GAIN_32 G = {0x85030000,1}; // Filter gain
libq_q31_t Yin,Yout;
libq_q63_t Y64;
Y64 = ( (pFilter->G.fracGain)*((libq_q63_t)Yin) )<<(pFilter->G.expGain+1);
Yout = Y64>>32;
```

1.9.8.7 HALF_L1_TO_L2_FACTOR Macro

C

```
#define HALF_L1_TO_L2_FACTOR 0x4716
```

Description

Conversion Factor from L1 to L2 Norms

Converts L1 norm (average absolute value) to L2 norm (RMS).

Remarks

For a sine wave the mean squared value is 1/2. So RMS = 1/sqrt(2). The average absolute value of a sine wave is 0.63661778. So the conversion factor from L1 to L2 norm is 0.70710678/0.63661778, which is 1.1107242. But since this number is bigger than one, it cannot be represented by a Q15 or Q31 constant, but 1.11107242/2 does fit into Q15 or Q31.

```
libq_q15_t    Ysample[MAX_NUM_SAMPLES];
libq_q15_t    avgAbsY;
libq_q15_t    pseudoYRMS;
libq_q16d15_t sumAbsY = 0;
libq_q31_t    temp32;
for (iSamp = 0; iSamp < nSamp; iSamp++)
{
    sumAbsY += abs(Ysample[iSamp]);
}
avgAbsY = sumAbsY/nSamp;
temp32 = (HALF_L1_TO_L2_FACTOR*avgAbsY)<<(1+1);
// <<1 because conversion factor is only half needed value
// other <<1 because integer multiply instead of fixed point multiply
pseudoYRMS = temp32>>16; // Q0.31 -> Q0.15
```


1.10 Files

Files

Name	Description
audio_equalizer.h	Audio Equalizer (DSP) functions for the PIC32MX and PIC32MZ device families
audio_equalizer_fixedpoint.h	Audio Equalizer (DSP) fixed point typedefs.
GraphicEqualizer6x2_Q15.h	16 Bit Filter definition for 6 Bands, with 2 Filters/Band.
GraphicEqualizer6x2_Q31.h	32 Bit Filter definition for 6 Bands, with 2 Filters/Band.
myFilters4x2_Q15.h	16 Bit Filter definition for 4 Bands, with 2 Filters/Band.
myFilters4x2_Q31.h	32 Bit Filter definition for 4 Bands, with 2 Filters/Band.
myFilters4x3_Q15.h	16 Bit Filter definition for 4 Bands, with 3 Filters/Band.
myFilters4x3_Q31.h	32 Bit Filter definition for 4 Bands, with 3 Filters/Band.
myFilters5x2_Q15.h	16 Bit Filter definition for 5 Bands, with 2 Filters/Band.
myFilters5x2_Q31.h	32 Bit Filter definition for 5 Bands, with 2 Filters/Band.
myFilters6x2_Q15.h	16 Bit Filter definition for 6 Bands, with 2 Filters/Band.
myFilters6x2_Q31.h	32 Bit Filter definition for 6 Bands, with 2 Filters/Band.
myFilters7x2_Q15.h	16 Bit Filter definition for 7 Bands, with 2 Filters/Band.
myFilters7x2_Q31.h	32 Bit Filter definition for 7 Bands, with 2 Filters/Band.
myFilters8x2_Q15.h	16 Bit Filter definition for 8 Bands, with 2 Filters/Band.
myFilters8x2_Q31.h	32 Bit Filter definition for 8 Bands, with 2 Filters/Band.
ParametricFilters1x8_Q15.h	16 Bit Filter definition for an 8 filter chain
ParametricFilters1x8_Q31.h	32 Bit Filter definition for an 8 filter chain
ParametricFilters1x8_Q31_Hacked.h	32 Bit Filter definition for an 8 filter chain, with edits to show 16 bit effects

1.10.1 audio_equalizer.h

Audio Equalizer Library

This [library](#) provides support for Audio band equalization filtering. Both band-specific and parametric equalization filtering is supported.

Except where noted functions are implemented in efficient assembly with C-callable prototypes. In some cases both 16-bit and 32-bit functions are supplied, providing the user with a choice of resolution and performance.

For most functions, input and output data is represented by 16-bit fractional numbers in Q15 format, which is the most commonly used data format for signal processing. Some functions use other data formats internally for increased precision of intermediate results.

The Q15 data type used by these functions is specified as `int16_t` in the C header file that is supplied with the [library](#). Note that within C code, care must be taken to avoid confusing fixed-point values with integers. To the C compiler, objects declared with `int16_t` type are integers, not fixed-point, and all arithmetic operations performed on those objects in C will be done as integers. Fixed-point values have been declared as `int16_t` only because the standard C language does not include intrinsic support for fixed-point data types.

Some functions also have versions operating on 32-bit fractional data in Q31 format. These functions operate similarly to their

16-bit counterparts.

Signed fixed point types are defined as follows:


$Q_n.m$ where:

- n is the number of data bits to the left of the radix point
- m is the number of data bits to the right of the radix point
- a signed bit is implied



















Unique variable types for fractional representation are also defined:














Exact Name # Bits Required Type Q0.15 (Q15) 16 [libq_q0d15_t](#) Q0.31 (Q31) 32 [libq_q0d31_t](#)

Enumerations



	Name	Description
	BAND_ENERGY_UNITS	Determines what units are used in reporting band energy.

Functions





	Name	Description
	AUDIO_EQUALIZER_BandEnergyGetQ15	Get band energy estimate for a given filter band. "Q15" suffix designates this routine is for signals with Q15 fixed point format.
	AUDIO_EQUALIZER_BandEnergyGetQ31	Get band energy estimate for a given filter band. "Q31" suffix designates this routine is for signals with Q31 fixed point format.
	AUDIO_EQUALIZER_BandEnergyNSamplesSet	Resets number of samples used to update band energy measurements.
	AUDIO_EQUALIZER_BandEnergySumsInit	Initialize band energy measurements, clearing band energy sum array and number of energy samples for each band.
	AUDIO_EQUALIZER_BandEnergyUpdateQ15	Update band energy estimate for a given filter band with new filter output. "Q15" suffix designates this routine is for signals with Q15 fixed point format.
	AUDIO_EQUALIZER_BandEnergyUpdateQ31	Update band energy estimate for a given filter band with new filter output. "Q31" suffix designates this routine is for signals with Q31 fixed point format.
	AUDIO_EQUALIZER_Cascade2inQ15	Performs a single output of a cascade of 2 biquad IIR filters.
	AUDIO_EQUALIZER_Cascade2inQ31	Performs a single output of a cascade of 2 biquad IIR filters.
	AUDIO_EQUALIZER_Cascade8inQ15	Performs a single output of a cascade of 8 biquad IIR filters.
	AUDIO_EQUALIZER_Cascade8inQ31	Performs a single output of a cascade of 8 biquad IIR filters.
	AUDIO_EQUALIZER_FilterGainAdjustQ15	Adjusts a filter gain structure by the integer gain adjustment provided
	AUDIO_EQUALIZER_FilterGainAdjustQ31	Adjusts a filter gain structure by the integer gain adjustment provided
	AUDIO_EQUALIZER_FilterGainGetQ15	Gets the filter gain for a given band and filter.
	AUDIO_EQUALIZER_FilterGainGetQ31	Gets the filter gain for a given band and filter.
	AUDIO_EQUALIZER_FilterGainSetQ15	Gets the filter gain for a given band and filter.
	AUDIO_EQUALIZER_FilterGainSetQ31	Gets the filter gain for a given band and filter.
	AUDIO_EQUALIZER_GainNormalizeQ15	Normalize all the EQUALIZER_FILTER_GAIN 's in a filter array so that the gains can be applied correctly by each filtering function.
	AUDIO_EQUALIZER_GainNormalizeQ31	Normalize all the EQUALIZER_FILTER_GAIN 's in a filter array so that the gains can be applied correctly by each filtering function.

	AUDIO_EQUALIZER_IIRinQ15	Applies equalization filter defined by *pFilter to Xin and provides single output.
	AUDIO_EQUALIZER_IIRinQ15andC	Applies equalization filter defined by *pFilter to Xin and provides single output.
	AUDIO_EQUALIZER_IIRinQ15FastC	Applies equalization filter defined by *pFilter to Xin and provides single output.
	AUDIO_EQUALIZER_IIRinQ31	Applies equalization filter defined by *pFilter to Xin and provides single output.
	AUDIO_EQUALIZER_IIRinQ31andC	Applies equalization filter defined by *pFilter to Xin and provides single output.
	AUDIO_EQUALIZER_Parallel4x2inQ15	Performs 4 parallel IIR filters, with 2 series biquad filters each, and sums the result.
	AUDIO_EQUALIZER_Parallel4x2inQ31	Performs 4 parallel IIR filters, 2 series biquad filters each, and sums the result.
	AUDIO_EQUALIZER_Parallel8x2inQ15	Performs 8 parallel IIR filters, with 2 series biquad filters each, and sums the result.
	AUDIO_EQUALIZER_Parallel8x2inQ31	Performs 8 parallel IIR filters, with 2 series biquad filters each, and sums the result.
	AUDIO_EQUALIZER_ParallelNx2inQ15	Performs N parallel IIR filters, 2 series biquad filters each, and sums the result.
	AUDIO_EQUALIZER_ParallelNx2inQ31	Performs N parallel IIR filters, 2 series biquad filters each, and sums the result.
	AUDIO_EQUALIZER_ParallelNxMinQ15	Performs N parallel IIR filters, M series biquad filters each, and sums the result.
	AUDIO_EQUALIZER_ParallelNxMinQ31	Performs N parallel IIR filters, M series biquad filters each, and sums the result.

Macros

	Name	Description
	AUDIO_EQUALIZER_MAX_NBANDS	Maximum number of filter bands supported.
	HALF_L1_TO_L2_FACTOR	Converts L1 norm (average absolute value) to L2 norm (RMS).

Structures

	Name	Description
	EQUALIZER_FILTER	Typedef for equalizer IIR filter definition structure.
	EQUALIZER_FILTER_32	Typedef for equalizer IIR filter definition structure.
	EQUALIZER_FILTER_GAIN	Typedef for equalizer filter gain structure.
	EQUALIZER_FILTER_GAIN_32	Typedef for equalizer filter gain structure.

File Name

audio_equalizer.h

Company

Microchip Technology Inc.

1.10.2 audio_equalizer_fixedpoint.h

Audio Equalizer Library Fixedpoint Typedefs

Signed fixed point types are defined as follows:










$Q_n.m$ where:

- n is the number of data bits to the left of the radix point
- m is the number of data bits to the right of the radix point
- a signed bit is implied

Unique variable types for fractional representation are also defined:

Exact Name # Bits Required Type Q0.15 (Q15) 16 [libq_q0d15_t](#) Q0.31 (Q31) 32 [libq_q0d31_t](#)

Types

	Name	Description
	libq_q0d15_t	Typedef for the Q0.15 fixed point data type.
	libq_q0d16_t	Typedef for the Q0.16 fixed point data type.
	libq_q0d31_t	Typedef for the Q0.31 fixed point data type.
	libq_q0d63_t	Typedef for the Q0.63 fixed point data type
	libq_q15_t	Typedef for the Q0.15 fixed point data type.
	libq_q15d16_t	Typedef for the Q15.16 fixed point data type
	libq_q16d15_t	Typedef for the Q16d15 fixed point data type
	libq_q31_t	Typedef for the Q0.31 fixed point data type.
	libq_q63_t	Typedef for the Q0.63 fixed point data type

File Name

audio_equalizer_fixedpoint.h

Company

Microchip Technology Inc.

1.10.3 GraphicEqualizer6x2_Q15.h

Audio Equalizer Library Filter Definition (Created by GraphicEqualizerDesign.m)

16 Bit Filter definitions for 6 Bands, with 2 Filters/Band. See the file GraphicEqualizer6x2_Q15.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.22603 (1.77003 dB)
 Unscaled Filter Response Average = 1.13514 (1.10099 dB)
 Scaled Filter Response Peak = 1 (3.85731e-15 dB)
 Scaled Filter Response Average = 0.925866 (-0.669039 dB)

File Name

GraphicEqualizer6x2_Q15.h

1.10.4 GraphicEqualizer6x2_Q31.h

Audio Equalizer Library Filter Definition (Created by GraphicEqualizerDesign.m)

32 Bit Filter definitions for 6 Bands, with 2 Filters/Band. See the file GraphicEqualizer6x2_Q31.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.22607 (1.77027 dB)
Unscaled Filter Response Average = 1.13523 (1.1017 dB)
Scaled Filter Response Peak = 1 (0 dB)
Scaled Filter Response Average = 0.925916 (-0.668572 dB)

File Name

GraphicEqualizer6x2_Q31.h

1.10.5 myFilters4x2_Q15.h

Audio Equalizer Library Filter Definition (Created by GraphicEqualizerFilterDesignScript.m)

16 Bit Filter definitions for 4 Bands, with 2 Filters/Band. See the file myFilters4x2_Q15.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.59502 (4.05533 dB)
Unscaled Filter Response Average = 1.4646 (3.31439 dB)
Scaled Filter Response Peak = 1 (-9.64327e-16 dB)
Scaled Filter Response Average = 0.918233 (-0.740944 dB)

File Name

myFilters4x2_Q15.h

1.10.6 myFilters4x2_Q31.h

Audio Equalizer Library Filter Definition (Created by GraphicEqualizerFilterDesignScript.m)

32 Bit Filter definitions for 4 Bands, with 2 Filters/Band. See the file myFilters4x2_Q31.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.59493 (4.05481 dB)
Unscaled Filter Response Average = 1.46456 (3.31414 dB)
Scaled Filter Response Peak = 1 (1.92865e-15 dB)
Scaled Filter Response Average = 0.918261 (-0.740674 dB)

File Name

myFilters4x2_Q31.h

1.10.7 myFilters4x3_Q15.h

Audio Equalizer Library Filter Definition (Created by GraphicEqualizerFilterDesignScript.m)

16 Bit Filter definitions for 4 Bands, with 3 Filters/Band. See the file myFilters4x3_Q15.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.19305 (1.53317 dB)

Unscaled Filter Response Average = 1.11311 (0.93074 dB)
Scaled Filter Response Peak = 1 (-9.64327e-16 dB)
Scaled Filter Response Average = 0.932993 (-0.602429 dB)

File Name

myFilters4x3_Q15.h

1.10.8 myFilters4x3_Q31.h

Audio Equalizer Library Filter Definition (Created by GraphicEqualizerFilterDesignScript.m)

32 Bit Filter definitions for 4 Bands, with 3 Filters/Band. See the file myFilters4x3_Q31.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.19305 (1.53315 dB)
Unscaled Filter Response Average = 1.1131 (0.930694 dB)
Scaled Filter Response Peak = 1 (-1.92865e-15 dB)
Scaled Filter Response Average = 0.93299 (-0.602457 dB)

File Name

myFilters4x3_Q31.h

1.10.9 myFilters5x2_Q15.h

Audio Equalizer Library Filter Definition (Created by GraphicEqualizerFilterDesignScript.m)

16 Bit Filter definitions for 5 Bands, with 2 Filters/Band. See the file myFilters5x2_Q15.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.58718 (4.01254 dB)
Unscaled Filter Response Average = 1.51086 (3.58446 dB)
Scaled Filter Response Peak = 1 (-1.92865e-15 dB)
Scaled Filter Response Average = 0.95191 (-0.42808 dB)

File Name

myFilters5x2_Q15.h

1.10.10 myFilters5x2_Q31.h

Audio Equalizer Library Filter Definition (Created by GraphicEqualizerFilterDesignScript.m)

32 Bit Filter definitions for 5 Bands, with 2 Filters/Band. See the file myFilters5x2_Q31.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.58728 (4.01309 dB)
Unscaled Filter Response Average = 1.51085 (3.58445 dB)
Scaled Filter Response Peak = 1 (1.92865e-15 dB)
Scaled Filter Response Average = 0.951849 (-0.428638 dB)

File Name

myFilters5x2_Q31.h

1.10.11 myFilters6x2_Q15.h

Audio Equalizer Library Filter Definition (Created by GraphicEqualizerFilterDesignScript.m)

16 Bit Filter definitions for 6 Bands, with 2 Filters/Band. See the file myFilters6x2_Q15.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.60219 (4.09428 dB)
Unscaled Filter Response Average = 1.53328 (3.71241 dB)
Scaled Filter Response Peak = 1 (-1.92865e-15 dB)
Scaled Filter Response Average = 0.956988 (-0.38187 dB)

File Name

myFilters6x2_Q15.h

1.10.12 myFilters6x2_Q31.h

Audio Equalizer Library Filter Definition (Created by GraphicEqualizerFilterDesignScript.m)

32 Bit Filter definitions for 6 Bands, with 2 Filters/Band. See the file myFilters6x2_Q31.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.6021 (4.09378 dB)
Unscaled Filter Response Average = 1.53332 (3.71267 dB)
Scaled Filter Response Peak = 1 (0 dB)
Scaled Filter Response Average = 0.957073 (-0.381102 dB)

File Name

myFilters6x2_Q31.h

1.10.13 myFilters7x2_Q15.h

Audio Equalizer Library Filter Definition (Created by GraphicEqualizerFilterDesignScript.m)

16 Bit Filter definitions for 7 Bands, with 2 Filters/Band. See the file myFilters7x2_Q15.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.58661 (4.00941 dB)
Unscaled Filter Response Average = 1.5339 (3.71595 dB)
Scaled Filter Response Peak = 1 (-2.89298e-15 dB)
Scaled Filter Response Average = 0.966778 (-0.293464 dB)

File Name

myFilters7x2_Q15.h

1.10.14 myFilters7x2_Q31.h

Audio Equalizer Library Filter Definition (Created by GraphicEqualizerFilterDesignScript.m)

32 Bit Filter definitions for 7 Bands, with 2 Filters/Band. See the file myFilters7x2_Q31.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.58659 (4.00928 dB)
Unscaled Filter Response Average = 1.53388 (3.71583 dB)
Scaled Filter Response Peak = 1 (1.92865e-15 dB)
Scaled Filter Response Average = 0.96678 (-0.293448 dB)

File Name

myFilters7x2_Q31.h

1.10.15 myFilters8x2_Q15.h

Audio Equalizer Library Filter Definition (Created by GraphicEqualizerFilterDesignScript.m)

16 Bit Filter definitions for 8 Bands, with 2 Filters/Band. See the file myFilters8x2_Q15.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.62333 (4.20811 dB)
Unscaled Filter Response Average = 1.49845 (3.51285 dB)
Scaled Filter Response Peak = 1 (-9.64327e-16 dB)
Scaled Filter Response Average = 0.923075 (-0.695262 dB)

File Name

myFilters8x2_Q15.h

1.10.16 myFilters8x2_Q31.h

Audio Equalizer Library Filter Definition (Created by GraphicEqualizerFilterDesignScript.m)

32 Bit Filter definitions for 8 Bands, with 2 Filters/Band. See the file myFilters8x2_Q31.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.62342 (4.2086 dB)
Unscaled Filter Response Average = 1.49842 (3.51269 dB)
Scaled Filter Response Peak = 1 (1.92865e-15 dB)
Scaled Filter Response Average = 0.923006 (-0.695909 dB)

File Name

myFilters8x2_Q31.h

1.10.17 ParametricFilters1x8_Q15.h

Audio Equalizer Library Filter Definition (Created by ParametricEqualizerDesign.m)

16 Bit Filter definitions for 1 Bands, with 8 Filters/Band. See the file ParametricFilters1x8_Q15.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.25582 (1.97854 dB)

Unscaled Filter Response Average = 0.768183 (-2.29071 dB)

File Name

ParametricFilters1x8_Q15.h

1.10.18 ParametricFilters1x8_Q31.h

Audio Equalizer Library Filter Definition (Created by ParametricEqualizerDesign.m)

32 Bit Filter definitions for 1 Bands, with 8 Filters/Band. See the file ParametricFilters1x8_Q31.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.27398 (2.10325 dB)

Unscaled Filter Response Average = 0.768016 (-2.29259 dB)

File Name

ParametricFilters1x8_Q31.h

1.10.19 ParametricFilters1x8_Q31_Hacked.h

Audio Equalizer Library Filter Definition (Created by ParametricEqualizerDesign.m, with additional edits)

32 Bit Filter definitions for 1 Bands, with 8 Filters/Band. See the file ParametricFilters1x8_Q31.mat for details of the filter design.

Remarks

Unscaled Filter Response Peak = 1.27398 (2.10325 dB)

Unscaled Filter Response Average = 0.768016 (-2.29259 dB)

Filter 1 and Filter 2 coefficients are 16 bits wide, scaled from Q15 to Q31. These filters should behave identically to the [ParametricFilters1x8_Q15.h](#), proving that the problem is with filter coefficient rounding rather than calculating filters in 16 bit fractional math instead of 32 bit.

File Name

[ParametricFilters1x8_Q31.h](#)

Index

A

A Warning About Stereo Filters 1-40
 Application Examples 1-21
 Audio Equalization Filtering Library 1-1
 audio_equalizer.h 1-92
 AUDIO_EQUALIZER_BandEnergyGetQ15 function 1-81
 AUDIO_EQUALIZER_BandEnergyGetQ31 function 1-82
 AUDIO_EQUALIZER_BandEnergyNSamplesSet function 1-77
 AUDIO_EQUALIZER_BandEnergySumsInit function 1-76
 AUDIO_EQUALIZER_BandEnergyUpdateQ15 function 1-78
 AUDIO_EQUALIZER_BandEnergyUpdateQ31 function 1-80
 AUDIO_EQUALIZER_Cascade2inQ15 function 1-49
 AUDIO_EQUALIZER_Cascade2inQ31 function 1-50
 AUDIO_EQUALIZER_Cascade8inQ15 function 1-52
 AUDIO_EQUALIZER_Cascade8inQ31 function 1-53
 AUDIO_EQUALIZER_FilterGainAdjustQ15 function 1-67
 AUDIO_EQUALIZER_FilterGainAdjustQ31 function 1-68
 AUDIO_EQUALIZER_FilterGainGetQ15 function 1-69
 AUDIO_EQUALIZER_FilterGainGetQ31 function 1-71
 AUDIO_EQUALIZER_FilterGainSetQ15 function 1-70
 AUDIO_EQUALIZER_FilterGainSetQ31 function 1-72
 audio_equalizer_fixedpoint.h 1-94
 AUDIO_EQUALIZER_GainNormalizeQ15 function 1-73
 AUDIO_EQUALIZER_GainNormalizeQ31 function 1-74
 AUDIO_EQUALIZER_IIRinQ15 function 1-47
 AUDIO_EQUALIZER_IIRinQ15andC function 1-45
 AUDIO_EQUALIZER_IIRinQ15FastC function 1-46
 AUDIO_EQUALIZER_IIRinQ31 function 1-48
 AUDIO_EQUALIZER_IIRinQ31andC function 1-46
 AUDIO_EQUALIZER_MAX_NBANDS macro 1-88
 AUDIO_EQUALIZER_Parallel4x2inQ15 function 1-55
 AUDIO_EQUALIZER_Parallel4x2inQ31 function 1-56
 AUDIO_EQUALIZER_Parallel8x2inQ15 function 1-58
 AUDIO_EQUALIZER_Parallel8x2inQ31 function 1-59
 AUDIO_EQUALIZER_ParallelNx2inQ15 function 1-61
 AUDIO_EQUALIZER_ParallelNx2inQ31 function 1-63

AUDIO_EQUALIZER_ParallelNxMinQ15 function 1-64
 AUDIO_EQUALIZER_ParallelNxMinQ31 function 1-66

B

BAND_ENERGY_UNITS enumeration 1-88

C

Configuring the Library 1-16
 Core Exception Handling 1-18

E

Equalization Filters 1-28
 EQUALIZER_FILTER structure 1-89
 EQUALIZER_FILTER_32 structure 1-89
 EQUALIZER_FILTER_GAIN structure 1-90
 EQUALIZER_FILTER_GAIN_32 structure 1-90
 Example Filter Definition Files 1-28

F

Files 1-92
 Filter Validation Tools 1-36
 Filtering Performance 1-19
 Fixed Point Data and Mathematics 1-16

G

Glossary of Terms 1-13
 Graphic Equalization Filter Design Tools 1-29
 GraphicEqualizer6x2_Q15.h 1-95
 GraphicEqualizer6x2_Q31.h 1-95

H

HALF_L1_TO_L2_FACTOR macro 1-91

I

Introduction 1-2

L

libq_q0d15_t type 1-84
libq_q0d16_t type 1-85
libq_q0d31_t type 1-85
libq_q0d63_t type 1-86
libq_q15_t type 1-84
libq_q15d16_t type 1-87
libq_q16d15_t type 1-87
libq_q31_t type 1-86
libq_q63_t type 1-86
Library Interface 1-43
Library Overview 1-6

M

Matlab/Octave 1-28
myFilters4x2_Q15.h 1-96
myFilters4x2_Q31.h 1-96
myFilters4x3_Q15.h 1-96
myFilters4x3_Q31.h 1-97
myFilters5x2_Q15.h 1-97
myFilters5x2_Q31.h 1-97
myFilters6x2_Q15.h 1-98
myFilters6x2_Q31.h 1-98
myFilters7x2_Q15.h 1-98
myFilters7x2_Q31.h 1-99
myFilters8x2_Q15.h 1-99
myFilters8x2_Q31.h 1-99

P

Parametric Equalization Filter Design 1-33
ParametricFilters1x8_Q15.h 1-100
ParametricFilters1x8_Q31.h 1-100
ParametricFilters1x8_Q31_Hacked.h 1-100

R

Release Notes 1-14
Resource Requirements 1-11

S

SW License Agreement 1-15

U

Using the Library 1-16