

NatureDSP Signal

Digital Signal Processing

Library Reference

Table of Contents

Preface	5
About this manual	5
Supported targets	5
Notations.....	5
Abbreviations	5
Chapter 1 General library organization	6
1.1 Headers	6
1.2 Static variables and usage of C standard libraries.....	6
1.3 Types.....	6
1.4 Fractional formats	6
1.5 Compiler requirements	7
1.6 Call conventions	7
1.7 Overflow control.....	7
1.8 Performance issues	7
1.9 Basic operations	7
1.10 Brief function list	8
Chapter 2 Reference.....	10
2.1 FIR filters and related functions	10
2.1.1 Block real FIR filter	10
2.1.2 Complex block FIR filter	10
2.1.3 Symmetrical block real FIR filter.....	11
2.1.4 Decimating block real FIR filter.....	11
2.1.5 Interpolating block real FIR filter.....	12
2.1.6 Single-sample real FIR filter	12
2.1.7 Convolution	13
2.1.8 Correlation.....	13
2.1.9 Autocorrelation	14
2.1.10 Raw single sample delayed LMS algorithm.....	14
2.1.11 Blockwise Adaptive LMS algorithm	15
2.2 IIR filters	16
2.2.1 Bi-quad complex block IIR.....	16
2.2.2 Bi-quad double precision IIR filter.....	17
2.2.3 Lattice block Real IIR	18
2.2.4 Lattice complex block IIR	19
2.2.5 Lattice double precision IIR filter	19
2.3 Vector mathematics	20
2.3.1 Vector dot product	20
2.3.2 Vector sum	21
2.3.3 Power of a vector	21
2.3.4 Vector scaling with saturation.....	21
2.3.5 Reciprocal on Q15 numbers.....	22
2.3.6 Division of Q15 numbers	22
2.3.7 Logarithm	23
2.3.8 Sine/cosine.....	23
2.3.9 Tangent	24
2.3.10 Full arctangent.....	24
2.3.11 Arctangent.....	25
2.3.12 Common exponent	25
2.4 Matrix operations	26
2.4.1 Matrix multiply	26

2.4.2	Matrix transpose.....	26
2.5	Fast Fourier Transforms	27
2.5.1	FFT on complex data	27
2.5.2	FFT on real data.....	27
2.5.3	Inverse FFT on complex data.....	28
2.5.4	Inverse FFT forming real data	28
Chapter 3	Test environment and examples	30
3.1	How to build and run tests	30
3.2	FIR tests	31
3.3	Matrix operations tests.....	31
3.4	Vector operations tests	32
3.5	IIR tests	32
3.6	FFT tests	32
Chapter 4	Customer support.....	33

About this manual

Welcome to the **NatureDSP Signal Processing Library**, or **NatureDSP Signal** or library for short. The library is a collection of number highly optimized DSP functions for the DSP targets.

This source code library includes C-callable functions (ANSI-C language compatible) for general signal processing (filtering, correlation, convolution), math and vector functions.

Supported targets

Library supports Tensilica ConnX D2 target only. Call **IntegrIT** to support more targets.

Notations

This document uses the following conventions:

- program listings, program examples, interactive displays, filenames, variables and another software elements are shown in a special typeface (Courier);
- tables use smaller fonts.

Abbreviations

4FSK	4-level frequency shift keying
API	Application program interface
DSP	Digital signal processing
FFT	Fast Fourier transform
FIR	Finite impulse response
IDE	Integrated development environment
IFFT	Inverse Fast Fourier transform
IIR	Infinite impulse response
IR	Impulse response
LMS	Least mean squares

Chapter 1 General library organization

1.1 Headers

NatureDSP_Signal library is supplied with number of header files

./include/NatureDSP_types.h	Declarations of basic data types and compiler autodetection
./include/NatureDSP_Math.h	Prototypes of basic operations (see below)
./include/NatureDSP_baseopXD2.h	Mapping of basic operations to ConnxD2 specific intrinsics
NatureDSP_Signal.h	Declarations of library functions

1.2 Static variables and usage of C standard libraries

Library does not have non-constant static variables and do not require special link-time auto initialization of variables before startup.

Library does not require floating point support from compiler and toolchain.

1.3 Types

Library uses standardized ANSI C types with defined length

int8_t	8-bit signed value
uint8_t	8-bit unsigned value
int16_t	16-bit signed value
uint16_t	16-bit unsigned value
int32_t	32-bit signed value
uint32_t	32-bit unsigned value

Additionally, it introduced special type `tDoubleWord` that represents the pair of two 32-bit numbers. It mostly used for data alignment.

1.4 Fractional formats

Unless specifically noted, library functions use Q_{15} format, or in another words, $Q_{0.15}$.

In a $Q_{m.n}$ format, there are m bits used to represent the two's complement integer portion of the number, and n bits used to represent the two's complement fractional portion. $m+n+1$ bits are needed to store a general $Q_{m.n}$ number. The extra bit is needed to store the sign of the number in the most-significant bit position. The representable integer range is specified by $[-2^m, 2^{m-1}]$ and the finest fractional resolution is 2^{-n} . Normally, m from Q notation is omitted (because total length is defined of data type used for operand) and it is simply written as Q_m .

For example, the most commonly used format is Q_{15} . Q_{15} means that a 16-bit word is used to express a signed number between positive and negative one. So, minimum value -32768 represents $-1.Q_{15}$ and maximum value 32767 represents $0,999969482421875Q_{15}$. Another typical cases used in the library are collected in the table below:

Format	Range	Minimum value	Maximum value
$Q_{0.15}$	-1 ... 0,999969	-32768	32767
$Q_{6.9}$	-64 ... 63,998	-32768	32767
$Q_{0.30}$	-2 ... 1,9999999991	-2147483648	2147483647
$Q_{1.30}$	-1 ... 0,9999999995	-2147483648	2147483647

The most-significant binary digit is interpreted as the sign bit in any Q format number. Thus, in Q15 format, the decimal point is placed immediately to the right of the sign bit. The fractional portion to the right of the sign bit is stored in regular two's complement format.

1.5 Compiler requirements

To enable normal compiler autodetection in the `NatureDSP_types.h` you have to define in the compiler options two special symbols via `-D` switch:

```
-DCONFIG_Connx_D2
-DPROC_Proc_D2
```

Normally, when you build up the using Eclipse environment it is done automatically by IDE.

1.6 Call conventions

Library uses ANSI-C call conventions.

1.7 Overflow control

If not especially noted, library does not check real dynamic range of input data so it is user's responsibility to select parameters and the scale of input data according to specific case. For example, putting to the `vec_power()` array of 4 24576 (0.75Q15) numbers will result overflow and you will receive 8192(0,25Q15) instead of right answer 73728 (2,25Q15).

The user is expected to conform to the range requirements if specified and take care to restrict the input range in such a way that the outputs do not overflow.

1.8 Performance issues

Real-time performance of all functions depends on fulfillment special restrictions applied to input arguments. Typically, for maximum performance, user have to use **aligned data arrays** for storing input and output arguments, number of data should be **divisible by 2 or 4** and should be **greater than 4**. Specific requirements are given for each function in its API description.

Data alignment may be achieved by several methods:

- placing the data into special data section and make alignment at the link-time
- dynamically allocate arrays of slighter bigger size and align pointers
- use operator union to overlap allocated arrays with array of `tDoubleWord` numbers (language C guarantees that all members in union will be aligned to the member requiring maximum alignment)

Test examples use that last method and you may use `ALIGNED_ARRAY()` macro to create aligned arrays of needed size.

1.9 Basic operations

Prototype	Fractional format conversion	Purpose
<code>int16_t S add ss (int16_t x, int16_t y)</code>	Q15 Q15->Q15	addition with saturation
<code>int32_t L add ss (int16_t x, int16_t y)</code>	Q15 Q15->Q31	
<code>int32_t L add ll (int32_t x, int32_t y)</code>	Q31 Q31->Q31	
<code>int16_t S sub ss (int16_t x, int16_t y)</code>	Q15 Q15->Q15	subtraction with saturation
<code>int32_t L sub ss (int16_t x, int16_t y)</code>	Q15 Q15->Q31	
<code>int32_t L sub ll (int32_t x, int32_t y)</code>	Q31 Q31->Q31	
<code>int16_t S neg s (int16_t x)</code>	Q15->Q15	negation with saturation
<code>int32_t L neg l (int32_t x)</code>	Q31->Q31	
<code>int16_t S abs s (int16_t x)</code>	Q15->Q15	absolute value with saturation
<code>int32_t L abs l (int32_t x)</code>	Q31->Q31	
<code>int16_t S_sature_l (int32_t x)</code>	Q16.15->Q15	saturation to 16-bit range

Prototype	Fractional format conversion	Purpose
int16_t S_extract_l (int32_t x)	Q31->Q15	extract higher 16 bits
int32_t L_deposit_s (int16_t x)	Q15->Q31	shift left by 16
int32_t L_round_l (int32_t x)	Q31->Q31	Rounding (add 0x8000) with saturation
int16_t S_round_l (int32_t x)	Q31->Q15	Rounding (add 0x8000) with saturation and shift left by 16
int16_t S_exp_l (int32_t x)	Q31	getting exponent returns 0 if x==0
int16_t S_exp_s (int16_t x)	Q15	getting exponent, returns 31 if x==0
int16_t S_exp0_l (int32_t x)	Q31	getting exponent, returns 15 if x==0
int16_t S_exp0_s (int16_t x)	Q15	getting exponent, returns 15 if x==0
int32_t L_shl_s (int16_t x, int16_t t)	Qx->Q(x+t-16)	shift left with saturation
int32_t L_shl_l (int32_t x, int16_t t)	Qx->Q(x+t)	
int16_t S_shl_s (int16_t x, int16_t t)	Qx->Q(x+t)	
int32_t L_shr_s (int16_t x, int16_t t)	Qx->Q(x-t-16)	shift right with saturation
int32_t L_shr_l (int32_t x, int16_t t)	Qx->Q(x-t)	
int16_t S_shr_s (int16_t x, int16_t t)	Qx->Q(x-t)	
int32_t L_mpy_ss (int16_t x, int16_t y)	Q15 Q15->Q31	fractional multiplication
int16_t S_mpy_ss (int16_t x, int16_t y)	Q15 Q15->Q15	
int32_t L_mpy_ls (int32_t x, int16_t y)	Q31 Q15->Q31	
int32_t L_mpy_ll (int32_t x, int32_t y)	Q31 Q31->Q31	
int32_t L_mul_ss (int16_t x, int16_t y)	Q15 Q15->Q30	multiplication : no saturation and overflow control
int16_t S_mul_ss (int16_t x, int16_t y)	Q15 Q15->Q14	
int32_t L_mul_ls (int32_t x, int16_t y)	Q31 Q15->Q30	
int32_t L_mul_ll (int32_t x, int32_t y)	Q31 Q31->Q30	
int32_t L_mac_ss (int32_t z, int16_t x, int16_t y)	Q15 Q15->Q31	MAC operations
int32_t L_mac_ls (int32_t z, int32_t x, int16_t y)	Q31 Q15->Q31	
int32_t L_mac_ll (int32_t z, int32_t x, int32_t y)	Q31 Q31->Q31	
int32_t L_mas_ss (int32_t z, int16_t x, int16_t y)	Q15 Q15->Q31	MAS operations
int32_t L_mas_ls (int32_t z, int32_t x, int16_t y)	Q31 Q15->Q31	
int32_t L_mas_ll (int32_t z, int32_t x, int32_t y)	Q31 Q31->Q31	
int16_t S_div_ls (int32_t x, int16_t y)	Q31 Q15 -> Q15	fast fractional division with saturation
int16_t S_div_ll (int32_t x, int32_t y)	Q31 Q31 -> Q15	
int32_t L_div_ll (int32_t x, int32_t y)	Q31 Q31 -> Q16.15	

1.10 Brief function list

Vectorized version	Scalar version	Purpose	Reference
FIR filters and related functions			
fir_bk		Block real FIR filter	2.1.1
fir_cbk		Complex block FIR filter	2.1.2
fir_sr		Symmetrical block real FIR filter	2.1.3
fir_dec		Decimating block real FIR filter	2.1.4
fir_interp		Interpolating block real FIR filter	2.1.5
fir_ss		Single-sample real FIR filter.	2.1.6
fir_convol		Convolution	2.1.7
fir_xcorr		Correlation	2.1.8
fir_acorr		Autocorrelation	2.1.9
fir_dlms		Raw single sample delayed LMS algorithm	2.1.10
fir_blms		Blockwise Adaptive LMS algorithm	2.1.11
IIR filters			
iir_bqc		Biquad Complex block IIR	2.2.1
iir_bqd		Biquad Double precision IIR filter	2.2.2
iir_latr		Lattice block Real IIR	2.2.3
iir_latc		Lattice complex block IIR	2.2.4
iir_latd		Lattice double precision IIR filter	2.2.5
Vector mathematics			
vec_dot		Vector dot product	2.3.1
vec_add		Vector sum	2.3.2

Vectorized version	Scalar version	Purpose	Reference
vec_power		Power of a vector	2.3.3
vec_shift		Vector scaling with saturation	2.3.4
vec_recip16	scl_recip16	Reciprocal on a vector of Q15 numbers	2.3.5
vec_divide	scl_divide	Division of Q15 numbers	2.3.6
vec_logn	scl_logn	Different kinds of logarithm	2.3.7
vec_log2	scl_log2		
vec_log10	scl_log10		
vec_sine	scl_sine	Sine	2.3.8
vec_cosine	scl_cosine	Cosine	
vec_tan	scl_tan	Tangent	2.3.9
vec_atan2	scl_atan2	Full arctangent	2.3.10
vec_atan	scl_atan	arctangent	2.3.11
vec_bexp	scl_bexp	Common exponent	2.3.12
Matrix operations			
mtx_mpy		Matrix multiply	2.4.1
mtx_trans		Matrix transpose	2.4.2
FFT			
fft_cplx		FFT on complex data	2.5.1
fft_real		FFT on real data	2.5.2
ifft_cplx		Inverse FFT on complex data	2.5.3
ifft_real		Inverse FFT forming real data	2.5.4

2.1 FIR filters and related functions

2.1.1 Block real FIR filter

Description Computes a real FIR filter (direct-form) using IR stored in vector *h*. The real data input is stored in vector *x*. The filter output result is stored in vector *r*. The filter calculates *N* output samples using *M* coefficients and requires last *M+N-1* samples on the input.

Algorithm

$$r_n = \sum_{m=0}^{M-1} h_{M-1-m} x_{n+m}, n = \overline{0 \dots N-1}$$

Prototype

```
void fir_bk (int16_t * restrict r,
            const int16_t * restrict x,
            const int16_t * restrict h,
            int N,
            int M)
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	N+M-1	input data. First in time corresponds to <i>x</i> [0], Q15
int16_t	h	M	filter coefficients in normal order, Q15
int	N		length of sample block
int	M		length of filter
Output:			
int16_t	r	N	output data, Q15

Returned value none

Restrictions *x*, *h*, *r* should not overlap

Performance *H* - aligned on a 4-bytes boundary

restrictions: *M* - a multiple of 2 exceeding 8

N - a multiple of 4 exceeding 8

2.1.2 Complex block FIR filter

Description Computes a complex FIR filter (direct-form) using complex IR stored in vector *h*. The complex data input is stored in vector *x*. The filter output result is stored in vector *r*. The filter calculates *N* output samples using *M* coefficients and requires last *M+N-1* samples on the input. Real and imaginary parts are interleaved and real parts go first (at even indexes).

Algorithm

$$r_n = \sum_{m=0}^{M-1} h_{M-1-m} x_{n+m}, n = \overline{0 \dots N-1}$$

Prototype

```
void fir_cbk (int16_t * restrict r,
             const int16_t * restrict x,
             const int16_t * restrict h,
             int N,
             int M)
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	2*(N+M-1)	input data. First in time corresponds to <i>x</i> [0], Q15
int16_t	h	2*M	filter coefficients in normal order, Q15
int	N		length of sample block
int	M		length of filter
Output:			
int16_t	r	2*N	output data, Q15

Returned value none
 Restrictions x, h, r should not overlap
 Performance x, h - aligned on a 4-bytes boundary
 restrictions: M - greater than eight

2.1.3 Symmetrical block real FIR filter

Description Computes a real FIR filter (direct-form) using half of IR stored in vector h . The real data input is stored in vector x . The filter output result is stored in vector r . The filter calculates N output samples using M coefficients and requires last $M+N-1$ samples on the input.

Algorithm

$$r_n = \sum_{m=0}^{M-1} h_{M-1-m} x_{n+m}, n = \overline{0 \dots N-1}$$

Prototype

```
void fir_sr (int16_t * restrict r,
            const int16_t * restrict x,
            const int16_t * restrict h,
            int N,
            int M)
```

Type	Name	Size	Description
Input:			
int16_t	x	N+M-1	input data. First in time corresponds to $x[0]$, Q15
int16_t	h	(M+1)/2	A half of filter coefficients, Q15. Last tap suggested to be a center of IR.
int	N		length of sample block
int	M		length of filter
Output:			
int16_t	r	N	output data, Q15

Returned value none
 Restrictions x, h, r should not be overlapped
 Performance x, h - aligned on a 4-bytes boundary
 restrictions: N, M - divisible by 4 and >8

2.1.4 Decimating block real FIR filter

Description Computes a real FIR filter (direct-form) with decimation using IR stored in vector h . The real data input is stored in vector x . The filter output result is stored in vector r . The filter calculates N output samples using M coefficients and requires last $D*N+M-1$ samples on the input.

NOTE:

To avoid aliasing IR should be synthesized in such a way to be narrower than input sample rate divided to $2D$.

Algorithm

$$r_n = \sum_{m=0}^{M-1} h_{M-1-m} x_{Dn+m}, n = \overline{0 \dots N-1}$$

Prototype

```
void fir_dec (int16_t * restrict r,
             const int16_t * restrict x,
             const int16_t * restrict h,
             int N,
             int M,
             int D)
```

Type	Name	Size	Description
Input:			
int16_t	x	D*N+M-1	input data. First in time corresponds to $x[0]$, Q15
int16_t	h	M	filter coefficients in normal order, Q15
int	N		length of sample block
int	M		length of filter
int	D		decimation factor
Output:			

	int16_t	r	N	output data, Q15
Returned value	none			
Restrictions	x, h, r should not be overlap D should exceed 1			
Performance restrictions:	x, h - aligned on a 4-bytes boundary N, M - divisible by 4 and >8 D - 2,3 or 4			

2.1.5 Interpolating block real FIR filter

Description Computes a real FIR filter (direct-form) with interpolation using IR stored in vector *h*. The real data input is stored in vector *x*. The filter output result is stored in vector *r*. The filter calculates *N*D* output samples using *M*D* coefficients and requires last *N+M*D-1* samples on the input. Function requires special storage format of IR. Original IR should be decimated by *D* with offsets *0..D-1*. Resulting *D* subfilters should be stored sequentially. For each subfilter the coefficient that corresponds to the newest sample is to be stored first.

Algorithm

$$r_{n-D+d} = \sum_{m=0}^{M-1} h_{(d+1)M-1-m} x_{n+m}, n = \overline{0..N-1}, d = \overline{0..D-1},$$

Prototype

```
void fir_interp (int16_t * restrict r,
                const int16_t * restrict x,
                const int16_t * restrict h,
                int N,
                int M,
                int D)
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	N+M*D-1	input data. First in time corresponds to x[0], Q15
int16_t	h	M*D	filter coefficients, Q15
int	N		length of sample block
int	M		length of subfilter
int	D		interpolation (upsample) factor
Output:			
int16_t	r	N*D	output data, Q15

Returned value none

Restrictions x, h, r should not be overlapped
D should be >1

Performance restrictions: x, h - aligned on a 4-bytes boundary
N - a multiple of 4 exceeding 8
M - a multiple of 2 exceeding 8
D - 2,3 or 4

2.1.6 Single-sample real FIR filter

Description Passes one sample via real FIR filter (direct-form) using IR stored in vector *h* and moves delay line with samples. The real data input (delay line) is stored in vector *x*. The filter output result is stored in vector *r*. The filter calculates one output samples using *M* coefficients and requires *M* samples on the input.

Algorithm

$$r = \sum_{m=0}^{M-1} h_{M-1-m} x_m$$

$$x_{m-1} = x_m, m = \overline{1..M-1}$$

Prototype

```
int32_t fir_ss (int16_t * restrict x,
                const int16_t * restrict h,
                int M)
```

Arguments

Type	Name	Size	Description
Input:			

int16_t	x	M	input data. First in time corresponds to $x[0]$, Q15. User should place new arrived sample to the last position of x .
int16_t	h	M	filter coefficients in normal order, Q15
int	M		length of filter
Output:			
int16_t	x	M	shifted in time input data by 1 sample

Returned value

int32_t	r		filtered data, Q30
---------	---	--	--------------------

Restrictions

x, h should not overlap

Performance

H - aligned on a 4-bytes boundary

restrictions:

M - a multiple of 4 exceeding 8

2.1.7 Convolution

Description

Performs convolution between vectors x and y . Algebraically, convolution is the same operation as multiplying the polynomials whose coefficients are the elements of x and y .

The convolution calculates $M+N-1$ output samples using vectors of size N and M correspondingly.

Function is equivalent to MATLAB's `conv()` function operating on real data.

Algorithm

$$r_{k+M-1} = \sum_{n=n_{\min}}^{n_{\max}} x_{n+k} y_{M-1-n}, k = -(M-1) \dots (N-1)$$

$$n_{\min} = \max(0, -k)$$

$$n_{\max} = \min(M-1, N-1-k)$$

Note: summation is performed over all values which lead to legal subscripts of x and y .

Prototype

```
void fir_convol (int16_t * restrict r,
                const int16_t * restrict x,
                const int16_t * restrict y,
                int N,
                int M)
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	N	input data, Q15
int16_t	y	M	input data, Q15
int	N		length of x
int	M		length of y
Output:			
int16_t	r	M+N-1	output data, Q15

Returned value

none

Restrictions

x, y, r should not overlap

Performance

x, y - aligned on a 4-bytes boundary

restrictions:

N, M - multiples of 4 exceeding 8

2.1.8 Correlation

Description

Estimates the cross-correlation between vectors x and y .

The cross-correlation calculates $M+N-1$ output samples using vectors of size N and M correspondingly.

Function is equivalent to MATLAB's `xcorr()` function operating on real data. It is also similar `fir_convol()` to but operates on reversed y sequence.

Algorithm

$$r_{k+M-1} = \sum_{n=n_{\min}}^{n_{\max}} x_{n+k} y_n, k = -(M-1) \dots (N-1)$$

$$n_{\min} = \max(0, -k)$$

$$n_{\max} = \min(M-1, N-1-k)$$

Note: summation is performed over all values which lead to legal subscripts of x and y .

Prototype

```
void fir_xcorr (int16_t * restrict r,
               const int16_t * restrict x,
               const int16_t * restrict y,
               int N,
               int M)
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	N	input data, Q15
int16_t	y	M	input data, Q15
int	N		length of x
int	M		length of y
Output:			
int16_t	r	M+N-1	output data, Q15

Returned value

none

Restrictions

x, y, r should not overlap

Performance

x, y - aligned on a 4-bytes boundary

restrictions:

N, M - multiples of 4 exceeding 8

2.1.9 Autocorrelation

Description

Estimates the auto-correlation between of vector x . Returns autocorrelation in a length N vector, where x is length N vector.

This function is similar to MATLAB's `xcorr()` function operating on real data with one argument. However, MATLAB `xcorr()` function always returns autocorrelation of length $2*N-1$ but it is centered and symmetrical. In contrary, `fir_acorr()` function returns only the half of autocorrelation result.

Algorithm

$$r_k = \sum_{n=0}^{N-1-k} x_{n+k} x_n, k = \overline{0 \dots (N-1)}$$

Prototype

```
void fir_acorr ( int16_t * restrict r,
                const int16_t * restrict x,
                int N)
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	N	input data, Q15
int	N		length of x
Output:			
int16_t	r	N	output data, Q15

Returned value

none

Restrictions

x, r should not overlap

Performance

x - aligned on a 4-bytes boundary

restrictions:

N - a multiple of 4 exceeding 8

2.1.10 Raw single sample delayed LMS algorithm

Description

The Least Mean Square Adaptive Filter computes an update of all N coefficients of IR by adding the weighted error times the inputs to the original coefficients. The input array includes the last N inputs followed by a new single sample input.

Algorithm

$$r = \sum_{m=0}^{N-1} h_{N-1-m} x_{m+1}$$

$$h_{N-1-m} = h_{N-1-m} + b \cdot x_m, m = \overline{0 \dots N-1}$$

NOTE: Computation of filter output is done using only higher 16-bit words of IR coefficients (in fact converting them to Q15). However, when it performs coefficient update, it uses full Q31 accuracy.

Prototype

```
int32_t fir_dlms
(
    int32_t * restrict h,
    const int16_t * restrict x,
    int16_t b,
    int N)
```

Arguments

Type	Name	Size	Description
Input:			
int32_t	h	N	impulse response, Q31
int16_t	x	N+1	data vector. x[0]..x[N-1] are the samples processed on the last N steps with x[0] being the first in time, while x[N] is the newest sample which is to be processed now
int16_t	b		scaled error from previous step (Q16). Normally, is formed as a difference of a learning sequence and the filter output. Scaling has to be done to take into account needed LMS step.
int	N		length of h
Output:			
int32_t	h	N	updated impulse response, Q31

Returned value r filter output in Q30.
 Restrictions x, h should not overlap
 Performance N - a multiple of 4 exceeding 8
 restrictions:

2.1.11 Blockwise Adaptive LMS algorithm

Description Blockwise LMS algorithm performs filtering of input samples, computation of error over a block of reference samples and makes blockwise update of IR to minimize the error output.

Algorithm includes FIR filtering, calculation of correlation between the error output and reference signal and IR taps update based on that correlation.

NOTE: this algorithm consumes less CPU cycles per block than single sample algorithm at similar convergence rate.

Algorithm

$$b = \frac{\mu}{norm}$$

$$e_n = r_n - \sum_{m=0}^{M-1} h_{M-1-m} x_{m+n}, n = \overline{0 \dots N-1}$$

$$h_{M-1-m} = h_{M-1-m} + b \cdot \sum_{n=0}^{N-1} e_n x_{n+m}, m = \overline{0 \dots M-1}$$

NOTE: Computation of filter output is done using only higher 16-bit words of IR coefficients (in fact converting them to Q15). However, when it performs coefficient update, it uses full Q31 accuracy.

Prototype

```
void fir_blms ( int16_t * restrict e,
               int32_t * restrict h,
               const int16_t * restrict r,
               const int16_t * restrict x,
               int32_t norm,
               int16_t mu,
               int N,
               int M)
```

Arguments

Type	Name	Size	Description
Input:			
int32_t	h	M	impulse response, Q31
int16_t	r	N	reference (near end) data vector. First in time value is in $r[0]$
int16_t	x	N+M-1	input (far end) data vector. First in time value is in $x[0]$
int32_t	norm		normalization factor: power of signal multiplied by N, Q31
int16_t	mu		adaptation coefficient (LMS step), Q15
int	N		length of data block
int	M		length of h
Output:			

int16_t	e	N	estimated error, Q15
int32_t	y	M	updated impulse response, Q31

Returned value none
Restrictions h, x, r, y, e - should not overlap
Performance r, h, e - aligned on 4-bytes boundaries
restrictions: N, M - multiples of 4 exceeding 8

2.2 IIR filters

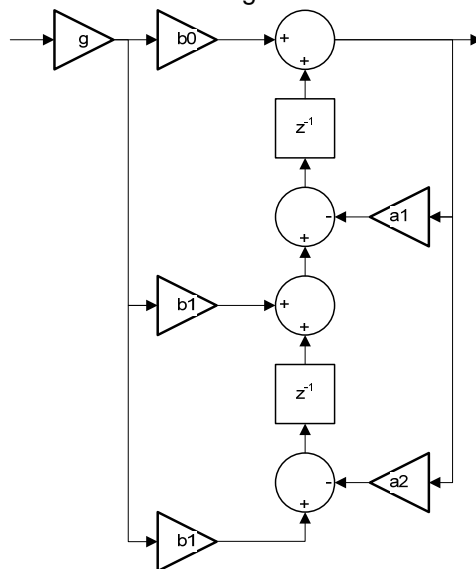
2.2.1 Bi-quad complex block IIR

Description Computes a complex IIR filter (cascaded IIR direct form II using 5 coefficients per bi-quad + gain term) . The complex data input are stored in vector x . The filter output result is stored in vector r . The filter calculates N output samples using SOS and G matrices.

NOTE:

Bi-quad coefficients may be derived from standard SOS and G matrices generated by MATLAB however its scale factors may require some tuning to find a compromise between quantization noise and possible overflows.

Algorithm Algorithm consists of applying sequentially M times bi-quad block filter with structure shown below and at the last stage scale factor g is formed from bi-quad scale factor and total gain



Prototype

```
void iir_bqc (
    int16_t * restrict r,
    int32_t * restrict d,
    const int16_t * restrict x,
    const int16_t * restrict coef,
    int16_t gain,
    int N, int M)
```

Arguments

Type	Name	Size	Description
Input:			
int32_t	d	4*M	Delay line elements from previous call. Should be zeroed prior to the first call
int16_t	x	2*N	input data. First in time corresponds to $x[0]$, Q15. Real and imaginary parts are interleaved and real part goes first.
int16_t	coef	M*6	filter coefficients written by blocks of 6 numbers: $g \ b0 \ b1 \ b2 \ a1 \ a2$. Fractional formats used: $b0 \ b1 \ b2 \ a1 \ a2$ Q14 g Q15
int16_t	gain		total gain, Q8

int	N		length of input vector (number of complex data points)
int	M		number of biquads
Output:			
int16_t	r	2*N	output complex data, Q15. Real and imaginary parts are interleaved and real part goes first.
int32_t	d	4*M	updated delay line

Returned value none
Restrictions $x, r, d, coef$ should not overlap
Performance restrictions:

2.2.2 Bi-quad double precision IIR filter

Description Computes a real IIR filter (cascaded IIR direct form II using 5 coefficients per bi-quad + gain term). The real data input are stored in vector x . The filter output result is stored in vector r . The filter calculates N output samples using SOS and G matrices.

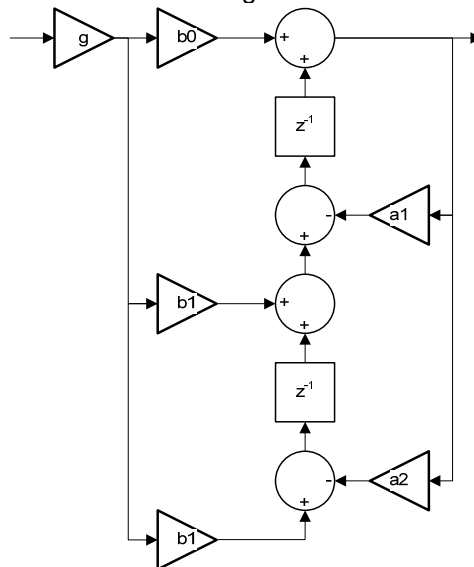
NOTE:

Bu-quad coefficients may be derived from standard SOS and G matrices generated by MATLAB however it scale factors may require some tuning to find a compromise between quantization noise and possible overflows.

This function is more accurate than `iir_bqc()` because it use double precision multiplications on the upper stage .

Algorithm

Algorithm consists of applying sequentially M times bi-quad block filter with structure shown below and at the last stage scale factor g is formed from bi-quad scale factor and total gain



Prototype

```
void iir_bqd (
    int16_t * restrict r,
    int32_t * restrict d,
    const int16_t * restrict x,
    const int16_t * restrict coef,
    int16_t gain,
    int N, int M)
```

Arguments

Type	Name	Size	Description
Input:			
int32_t	d	2*M	Delay line elements from previous call. Should be zeroed prior to the first call
int16_t	x	N	input data. First in time corresponds to $x[0]$, Q15.
int16_t	coef	M*6	filter coefficients written by blocks of 6

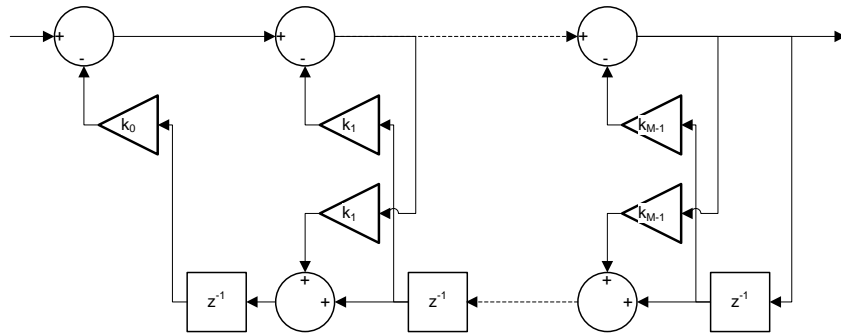
			numbers: g b0 b1 b2 a1 a2. Fractional formats used: b0 b1 b2 a1 a2 Q14 g Q15
int16_t	gain		total gain, Q8
int	N		length of sample block
int	M		number of biquads
Output:			
int16_t	r	N	output data, Q15
int32_t	d	2*M	updated delay line

Returned value none
Restrictions $x, r, d, coef$ should not overlap
Performance restrictions:

2.2.3 Lattice block Real IIR

Description Computes a real cascaded lattice autoregressive IIR filter using reflection coefficients stored in vector k . The real data input are stored in vector x . The filter output result is stored in vector r .

Algorithm Algorithm consists of applying sequentially M times IIR sections with structure shown below



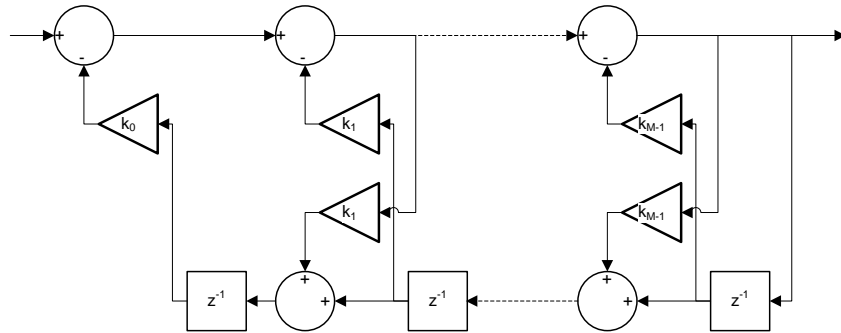
Prototype

```
void iir_latr
(
    int16_t *restrict r,
    int16_t *restrict b,
    const int16_t * restrict x,
    const int16_t * restrict k,
    int N, int M
)
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	b	M+1	Delay line elements from previous call. Should be zeroed prior to the first call
int16_t	x	N	input data. First in time corresponds to $x[0]$, Q15
int16_t	k	M	reflection coefficients, Q15 format
int	N		length of sample block
int	M		number of sections
Output:			
int16_t	r	N	output real data, Q15.
int16_t	b	M+1	updated delay line

Returned value none
Restrictions x, r, b, k should not be overlapped
Performance restrictions: b, k - aligned on 4-bytes boundaries
N - a multiple of 2
M - from the range 1...6



Prototype

```
void iir_latd
(
    int16_t *r,
    int32_t *restrict b,
    const int16_t * restrict x,
    const int16_t * restrict refl,
    int N, int M
)
```

Arguments

Type	Name	Size	Description
Input:			
int32_t	b	M+1	Delay line elements from previous call. Should be zeroed prior to the first call
int16_t	x	N	input data. First in time corresponds to x[0], Q15
int16_t	k	M	reflection coefficients, Q15 format
int	N		length of sample block
int	M		number of sections
Output:			
int16_t	r	N	output real data, Q15.
int32_t	b	M+1	updated delay line

Returned value none

Restrictions

x, r, b, k should not be overlapped

Performance

k - aligned on a 4-bytes boundary

restrictions:

N - a multiple of 4

M - from the range 1...6

2.3 Vector mathematics

2.3.1 Vector dot product

Description

This routine takes two vectors and calculates their dot product. The inputs are 16-bit short data and the output is a 32-bit number.

Algorithm

$$r = \sum_{n=0}^{N-1} x_n y_n$$

Prototype

```
int32_t vec_dot ( const int16_t * restrict x,
                 const int16_t * restrict y,
                 int N)
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	N	input data, Q15
int16_t	y	N	input data, Q15
int	N		length of vectors

Returned value

dot product r of all data pairs, Q30

Restrictions

none

Performance

none

restrictions:

2.3.2 Vector sum

Description This routine makes pair wise summation of vectors.

Algorithm $z_n = x_n + y_n, n = 0 \dots N - 1$

Prototype

```
void vec_add (int16_t * restrict z,
             const int16_t * restrict x,
             const int16_t * restrict y,
             int N)
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	N	input data, Q15
int16_t	y	N	input data, Q15
int	N		length of vectors
Output:			
int16_t	z	N	result, Q15

Returned value dot product r of all data pairs, Q30

Restrictions x, y, z - should not be overlapped

Performance none
restrictions:

2.3.3 Power of a vector

Description This routine computes power of vector.

Algorithm $r = \sum_{n=0}^{N-1} |x_n|^2$

Prototype

```
int32_t vec_power ( const int16_t * restrict x,
                   int N)
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	N	input data, Q15
int	N		length of vector

Returned value Sum of squares of a vector, Q30

Restrictions none

Performance none
restrictions:

2.3.4 Vector scaling with saturation

Description This routine makes shift with saturation of data values in the vector by given scale factor (degree of 2).

Algorithm $r_n = x_n \cdot 2^t$

Prototype

```
void vec_shift (          int16_t * restrict y,
                    const int16_t * restrict x,
                    int t,
                    int N);
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	N	input data, Q15
int	t		shift count. If positive, it shifts left with saturation, if negative it shifts right
int	N		length of vector
Output:			
int16_t	y	N	output data, Q15

Returned value none

Restrictions arrays should not overlap

Due to software pipeline the algorithm may speculatively read $x[N]$

Performance none
restrictions:

2.3.5 Reciprocal on Q15 numbers

Description This routine returns the fractional and exponential portion of the reciprocal of an vector *x* of Q15 numbers. Since the reciprocal is always greater than 1, it returns fractional portion *frac* in Q(15-*exp*) format and exponent *exp* so true reciprocal value in the Q16.15 may be found by shifting fractional part left by exponent value. For a reciprocal of 0, the result is not defined (but still be close to the maximum representative number)

Exponent value is always in range 1...16.

Accuracy is 1LSB of fractional part

Algorithm $frac_n \cdot 2^{exp_n} = 1/x_n, n = 0 \dots N-1$

Prototype

```
void vec_recip16 (
    int16_t * restrict frac,
    int16_t *exp,
    const int16_t * restrict x,
    int N)
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	N	input data, Q15
int	N		length of vectors
Output:			
int16_t	frac	N	fractional part of result, Q(15- <i>exp</i>)
int16_t	exp	N	exponent of result (1...16)

Returned value None

Restrictions *x*, *frac*, *exp* should not overlap

Due to software pipeline the algorithm speculatively reads entries *x*[*N*] and *x*[*N*+1]

Performance restrictions: None

Scalar version

Prototype

```
uint32_t scl_recip16 (int16_t x)
```

Arguments

Type	Name	Description
Input:		
int16_t	x	input data, Q15

Returned value packed value where exponent is in higher 16-bit part and fractional part is in the lower 16-bit part

2.3.6 Division of Q15 numbers

Description This routine performs pair wise division of vectors written in Q15 format. It returns the fractional and exponential portion of the division result. Since the division may generate result greater than 1, it returns fractional portion *frac* in Q(15-*exp*) format and exponent *exp* so true division result in the Q16.15 may be found by shifting fractional part left by exponent value (using `L_shl_1()` is necessary for that to saturate properly).

For division to 0, the result is not defined

Accuracy is 1LSB of fractional part

Algorithm $frac_n \cdot 2^{exp_n} = x_n / y_n, n = 0 \dots N-1$

Prototype

```
void vec_divide (int16_t * restrict frac,
    int16_t *exp,
    const int16_t * restrict x,
    const int16_t * restrict y, int N);
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	N	nominator, Q15
int16_t	y	N	denominator, Q15
int	N		length of vectors
Output:			
int16_t	frac	N	fractional parts of result, Q(15- <i>exp</i>)
int16_t	exp	N	exponents of result

Returned value none
 Restrictions $x, y, frac, exp$ should not be overlapping
 Due to software pipeline the algorithm speculatively reads $x[N]$
 Performance restrictions: none

Scalar version
 Prototype `uint32_t scl_divide (int16_t x, int16_t y)`

Type	Name	Description
Input:		
int16_t	x	nominator, Q15
int16_t	y	denominator, Q15

Returned value packed value where exponent is in higher 16-bit part and fractional part is in the lower 16-bit part

2.3.7 Logarithm

Description Different kinds of logarithm (base 2, natural, base 10). Results are represented in Q9 format or 0x8000 is returned on negative of zero argument.

This means, particularly, that
 $\log_2(1) \rightarrow 512$
 $\log_2(2^{30}) \rightarrow 512 \cdot 30$
 Accuracy 1 LSB worst case.

Algorithm $z_n = \log_K x_n, n = 0 \dots N-1, K = 2, e, 10$

Prototypes

```
void vec_log2 (          int16_t * restrict y,
                    const int32_t * restrict x,
                    int N)
void vec_logn (          int16_t * restrict y,
                    const int32_t * restrict x,
                    int N)
void vec_log10(          int16_t * restrict y,
                    const int32_t * restrict x,
                    int N)
```

Type	Name	Size	Description
Input:			
int32_t	x	N	input data, Q0
int	N		length of vectors
Output:			
int16_t	y	N	result, Q9

Returned value none
 Restrictions x, y should not overlap
 Due to software pipeline the algorithm speculatively reads entries $x[N]$ and $x[N+1]$

Performance restrictions: none

Scalar versions
 Prototypes

```
int16_t scl_log2 (int32_t x)
int16_t scl_log (int32_t x)
int16_t scl_log10(int32_t x)
```

Type	Name	Description
Input:		
int32_t	x	input data, Q0

Returned value result, Q9

2.3.8 Sine/cosine

Description Calculates $\sin(\pi \cdot x)$ or $\cos(\pi \cdot x)$ for number written in Q15 format. Returns result in Q15 format as well.

Accuracy $6.1 \cdot 10^{-5}$ Q15 worst case.

Algorithm $z_n = \sin(\pi x_n), n = 0 \dots N-1$ or

$z_n = \cos(\pi x_n), n = 0 \dots N-1$

Prototypes

```
void vec_sine (int16_t * restrict y,
              const int16_t * restrict x,
```

```

int N)
void vec_cosine (int16_t * restrict y,
                const int16_t * restrict x,
                int N)

```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	N	input data, Q15
int	N		length of vectors
Output:			
int16_t	y	N	result, Q15

Returned value

none

Restrictions

x, y should not overlap

Due to software pipeline the algorithm may speculatively read entries x[N] and x[N+1]

Performance

x, y - shall be aligned on a 4-bytes boundary

restrictions:

N - a multiple of 2

Scalar versions

Prototypes

```

int16_t scl_sine (int16_t x)
int16_t scl_cosine (int16_t x)

```

Arguments

Type	Name	Description
Input:		
int16_t	x	input data, Q15

Returned value

result, Q15

2.3.9 Tangent

Description

Calculates $\tan(\pi \cdot x)$ for number written in Q15 format. Returns result in Q16.15 format.

Absolute accuracy $6.1 \cdot 10^{-5}$ for $\pi \cdot x = (-\pi/4; \pi/4)$, $(-\pi \cdot 5/4; \pi \cdot 7/4)$

Relative accuracy $9.2 \cdot 10^{-5}$ for other values

Algorithm

$$z_n = \tan(\pi x_n), n = 0 \dots N - 1$$

Prototype

```

void vec_tan (      int32_t * restrict y,
                  const int16_t * restrict x,
                  int N)

```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	N	input data, Q15
int	N		length of vectors
Output:			
int32_t	y	N	result, Q16.15

Returned value

none

Restrictions

x, y should not overlap

Performance

none

restrictions:

Scalar versions

Prototype

```

int32_t scl_tan (int16_t x)

```

Arguments

Type	Name	Description
Input:		
int16_t	x	input data, Q15

Returned value

result, Q16.15

2.3.10 Full arctangent

Description

Function calculates four quadrant arctangent of complex numbers $\text{atan2}(x)/\pi$. Returns result in Q15 format.

Absolute phase accuracy 1 LSB

If both arguments are zero it returns zero as well.

Algorithm

$$z_n = \arg(x_n) / \pi, n = 0 \dots N - 1$$

Prototype

```

void vec_atan2 (int16_t* restrict z,
               const int16_t* restrict x,

```

		int N)		
Arguments	Type	Name	Size	Description
	Input:			
	int16_t	x	2*N	input complex data. Real and imaginary parts are interleaved and real part goes first.
	int	N		length of vectors
	Output:			
	int32_t	y	N	result, Q15

Returned value none
Restrictions x, y should not overlap
Due to software pipeline the algorithm may speculatively read entries $x[2*N]$, $x[2*N+1]$, $x[2*N+2]$, $x[2*N+3]$

Performance restrictions: N - a multiple of two

Scalar versions

Prototype

```
int16_t scl_atan2 (int16_t re,int16_t im)
```

Arguments

Type	Name	Description
Input:		
int16_t	re	real part of input
int16_t	im	imaginary part of input

Returned value result, Q15

2.3.11 Arctangent

Description Function calculates arctangent of number written in Q15 format. It scales output to pi so it is always in range -0x2000 ... 0x2000 which corresponds to the real phases $\pm\pi/4$

Absolute phase accuracy 1 LSB

It works faster than `vec_atan2()` so it have to be used when full phase coverage not needed.

Algorithm

$$z_n = \arctan(x_n) / \pi, n = 0 \dots N - 1$$

Prototype

```
void vec_atan (int16_t* restrict z,
              const int16_t* restrict x,
              int N )
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	N	input data, Q15
int	N		length of vectors
Output:			
int32_t	y	N	result, Q15

Returned value none

Restrictions x, y should not be overlapping
Due to software pipeline the algorithm may speculatively read entries $x[N]$, $x[N+1]$, $x[N+2]$, $x[N+3]$

Performance restrictions: N - multiple of 2

Scalar versions

Prototype

```
int16_t scl_atan (int16_t x)
```

Arguments

Type	Name	Description
Input:		
int16_t	x	input data, Q15

Returned value result, Q15

2.3.12 Common exponent

Description Computes the exponents (number of extra sign bits) of all values in the input vector and returns the minimum exponent. This will be useful in determining the maximum shift value that may be used in scaling a block of data.

Algorithm

$$z_n = \min(\overline{\text{norm}(x_n)})_{n=0..N-1}$$

where norm is exponent value (maximum possible shift count) for 32-bit data.

Prototype Arguments

int vec_bexp (const int16_t * restrict x, int N)

Type	Name	Size	Description
Input:			
int16_t	x	N	input data, Q15
int	N		length of vectors

Returned value

result – minimum exponent

Restrictions

none

Performance

none

restrictions:

Scalar versions

Prototype

int scl_bexp (int16_t x)

Arguments

Type	Name	Description
Input:		
int16_t	x	input data, Q15

Returned value

result

NOTE: this function is equivalent to s_exp0_1() operation from basic operation subset.

2.4 Matrix operations

2.4.1 Matrix multiply

Description

This function computes the expression $z = x * y$ for the matrices x and y . The columnar dimension of x must match the row dimension of y . The resulting matrix has the same number of rows as x and the same number of columns as y .

Algorithm

$$z_{n,p} = \sum_{m=0}^{M-1} x_{n,m} \cdot y_{m,p}, n = \overline{0..N-1}, p = \overline{0..P-1}$$

Prototype

int vec_bexp (const int16_t * restrict x, int N)

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	N*M	input matrix
int16_t	y	M*P	input matrix
int	N		number of rows in vectors x and z
int	M		number of columns in vector x and number of rows in vector y
int	P		number of columns in vectors y and z
Output:			
int16_t	z	N*P	output matrix

Returned value

none

Restrictions

arrays should not overlap

Due to software pipeline the algorithm may speculatively read entries $x[N*M]$, $x[N*M+1]$ and $y[M*P]..y[M*P + P-1]$

Performance

x, y - aligned on a 4-bytes boundary

restrictions:

M, P - multiples of 2

2.4.2 Matrix transpose

Description

This function transposes matrix.

Algorithm

$$y_{m,n} = x_{n,m}, n = \overline{0..N-1}, m = \overline{0..M-1}$$

Prototype

int vec_bexp (const int16_t * restrict x, int N)

Arguments

Type	Name	Size	Description
Input:			

int16_t	x	N*M	input matrix
int	N		number of rows in vector x and number of columns in vector y
int	M		number of columns in vector x and number of rows in vector y
Output:			
int16_t	y	M*N	output matrix

Returned value none

Restrictions arrays should not be overlapping
Due to software pipeline the algorithm may speculatively read entries $x[N*M]..x[N*M + 2*M-1]$.

Performance restrictions: The function is most efficient if M, N are multiples of two and the input array ($N > M$) or the output array ($N \leq M$) is aligned on a 4-bytes boundary.

2.5 Fast Fourier Transforms

2.5.1 FFT on complex data

Description This function makes FFT on complex data.
NOTES:
1. Bit-reversing permutation is done here.
2. FFT does not make scaling of input data and it should be done externally to avoid possible overflows.
3. FFT runs in-place algorithm so **INPUT DATA WILL BE DAMAGED** after the call

Algorithm $y = FFT(x)$

Prototype

```
void fft_cplx(
    int16_t* y,
    int16_t* x,
    int N)
```

Type	Name	Size	Description
Input:			
int16_t	x	2*N	input signal, Q15. Real and imaginary data are interleaved and real data goes first
int	N		FFT size
Output:			
int16_t	y	2*N	output spectrum, Q15

Returned value none

Restrictions arrays should not be overlapping
 x, y - should be aligned by 8 bytes
 N - may be 16, 32, 64, 128, 256 or 512.

Performance restrictions: none

2.5.2 FFT on real data

Description This function makes FFT on real data forming half of spectrum
NOTES:
1. Bit-reversing reordering is done here.
2. FFT does not make scaling of input data and it should be done externally to avoid possible overflows.
3. FFT runs in-place algorithm so **INPUT DATA WILL BE DAMAGED** after the call

Algorithm $y = FFT(real(x))$

Prototype

```
void fft_real(
    int16_t* y,
    int16_t* x,
    int N)
```

Type	Name	Size	Description
Input:			
int16_t	x	N	input signal, Q15. Real and imaginary data

			are interleaved and real data goes first
int	N		FFT size
Output:			
int16_t	y	(N/2+1)*2	output spectrum (positive side), Q15

Returned value none
Restrictions arrays should not be overlapping
x, y - should be aligned by 8 bytes
N - should be 32, 64, 128, 256, 512 or 1024.
Performance restrictions: none

2.5.3 Inverse FFT on complex data

Description This function makes inverse FFT on complex data.
NOTES:
1. Bit-reversing reordering is done here.
2. FFT does not make scaling of input data and it should be done externally to avoid possible overflows.
3. FFT runs in-place algorithm so **INPUT DATA WILL BE DAMAGED** after call

Algorithm $y = FFT^{-1}(x)$

Prototype

```
void ifft_cplx(
    int16_t* y,
    int16_t* x,
    int N)
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	2*N	input spectrum, Q15. Real and imaginary data are interleaved and real data goes first
int	N		FFT size
Output:			
int16_t	y	2*N	output signal, Q15. Real and imaginary data are interleaved and real data goes first

Returned value none
Restrictions arrays should not be overlapping
x, y - should be aligned by 8 bytes
N - should be 16, 32, 64, 128, 256 or 512.
Performance restrictions: none

2.5.4 Inverse FFT forming real data

Description This function makes inverse FFT on half spectral data forming real data samples
NOTES:
1. Bit-reversing reordering is done here.
2. IFFT does not make scaling of input data and it should be done externally to avoid possible overflows.
3. IFFT runs in-place algorithm so **INPUT DATA WILL BE DAMAGED** after call

Algorithm $y = real(FFT^{-1}(x))$

Prototype

```
void ifft_real(
    int16_t* y,
    int16_t* x,
    int N)
```

Arguments

Type	Name	Size	Description
Input:			
int16_t	x	(N/2+1)*2	input spectrum, Q15. Real and imaginary data are interleaved and real data goes first
int	N		FFT size
Output:			
int16_t	y	N	real output signal, Q15.

Returned value none

Restrictions

arrays should not be overlapping

x, y - should be aligned by 8 bytes

N - should be 32, 64, 128, 256, 512 or 1024.

Performance
restrictions:

None

3.1 How to build and run tests

It is assumed that you have downloaded and unpacked the NatureDSP Signal library delivery. For your convenience, the `<ROOT_DIR>` stands hereafter for the root directory of the delivery. It is also assumed that Xtensa building tools with support for the ConnX D2 core are installed and properly configured as specified by appropriate documents from Tensilica.

3.1.1 Building the NatureDSP Signal library

In order to build the object library for Tensilica ConnX D2 core, perform the following steps:

1. Switch to the library make file location:
`<ROOT_DIR>/src/NatureDSP_Signal/NatureDSP_Signal_xd2/makefile`
2. Open the `makefile` for editing and locate the following line:
`XTENSA_INSTALL = c:/xtensa/XtDevTools/install`
Adjust the path according to your installation of Xtensa tools and save the make file.
3. Though the library make file defines a few variants of building actions and options sets, you can just issue a `make` command with no arguments to launch the building process. Please refer to the list of available options described in the comments block of the make file.
4. After the make utility finishes, switch to the object library location to verify that it is up-to-date:
`<ROOT_DIR>/src/Libs/NatureDSP_Signal_xd2.a`

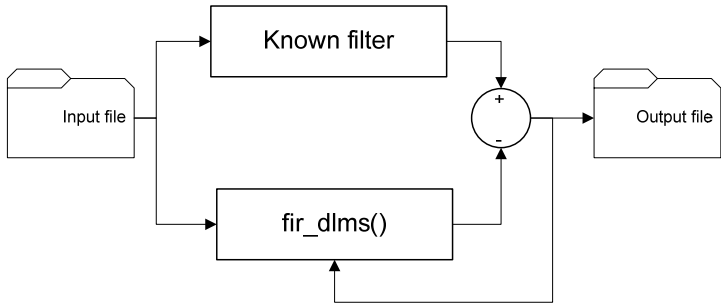
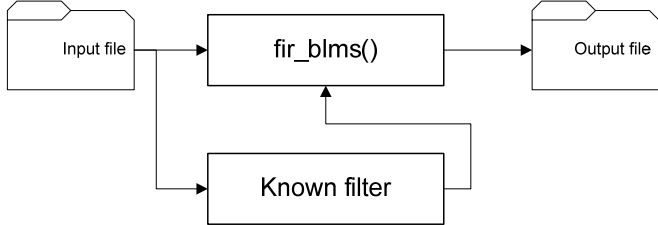
3.1.2 Building and running tests

Below are the steps needed to build and run the NatureDSP Signal library test:

1. Switch to the NatureDSP Signal object library file location and verify that it is up-to-date, otherwise refer to 3.1.1 for instruction on how to build it:
`<ROOT_DIR>/src/Libs/NatureDSP_Signal_xd2.a`
2. Switch to the library tests make file location:
`<ROOT_DIR>/src/test_xd2/makefile`
3. Open the `makefile` for editing and locate the following line:
`XTENSA_INSTALL = c:/xtensa/XtDevTools/install`
Adjust the path according to your installation of Xtensa tools and save the make file.
4. Issue a `make test` command to build the test executable and immediately run it using the instruction set simulator. Please note that the full test may take more than an hour before it finishes. Normally you should observe the list of completed tests and the name of the test in progress during this time.
5. After all the tests have been run and the program terminates you can find out the cycle count for a particular function from a text file containing the profiling results:
`<ROOT_DIR>/src/test_xd2/test_xd2.txt`

3.2 FIR tests

All test routines for FIR functions are collected in the file `test_fir.c`.

Subroutine	Test procedure
<code>testbkfir</code>	Tests function <code>fir_bk()</code> . Passes the signal read from file by blocks of 80 samples through the low pass filter of length 256 and write filtered output to the file.
<code>testfirss</code>	Tests function <code>fir_ss()</code> . Passes the signal read from file sample by sample through the low pass filter of length 256 and write filtered output to the file.
<code>testsrfir</code>	Tests function <code>fir_sr()</code> . Passes the signal read from file by blocks of 80 samples through the low pass filter of length 256 and write filtered output to the file.
<code>testfirinterp</code>	Tests function <code>fir_interp()</code> . Passes the signal read from file by blocks of 80 samples through the low pass filter of length 408 and write filtered interpolated output to the file. It is done for interpolation factor 2, 3 and 4.
<code>testfirdec</code>	Tests function <code>fir_dec()</code> . Passes the signal read from file by blocks of 80 samples through the low pass filter of length 256 and write filtered decimated output to the file. It is done for decimation factor 2, 3 and 4.
<code>test_fir_convol</code>	Tests function <code>fir_convol()</code> . Convolves 2 vectors of 80 and 56 samples and compare results with precalculated data.
<code>test_fir_xcorr</code>	Tests function <code>fir_xcorr()</code> . Correlates 2 vectors of 80 and 56 samples and compare results with precalculated data.
<code>test_fir_acorr</code>	Tests function <code>fir_acorr()</code> . Finds autocorrelation of vectors of 56 samples and compare results with precalculated data.
<code>testdlms</code>	<p>Tests function <code>fir_dlms()</code>. Reads signal (4FSK modulation sequence from signal space $[-3*2048 \ -1*2048 \ 1*2048 \ 3*2048]$), passes it through the filter with known IR and adapts another filter to minimize difference between their outputs:</p> 
<code>testblms</code>	<p>Tests function <code>fir_blms()</code>. Reads signal (4FSK modulation sequence from signal space $[-3*2048 \ -1*2048 \ 1*2048 \ 3*2048]$), passes it through the filter with known IR and adapts another filter to minimize difference between their outputs:</p> 

3.3 Matrix operations tests

All test routines for matrix functions are collected in the file `test_mtx.c`.

Subroutine	Test procedure
<code>test_mtx_mpy</code>	Tests function <code>mtx_mpy()</code> . Performs matrix multiplication (5x10 to 10x6 forming 5x6 result) and compares results with precalculated data.

Subroutine	Test procedure
test_mtx_trans	Tests function <code>mtx_trans()</code> . Performs matrix transpose and compares results with precalculated data.

3.4 Vector operations tests

All test routines for matrix functions are collected in the file `test_vec.c`.

Subroutine	Test procedure
test_vec_dot	Tests function <code>vec_dot()</code> . Computes dot product of test vectors and compares results with precalculated data.
test_vec_power	Tests function <code>vec_power()</code> . Computes power of test vectors and compares results with precalculated data.
test_vec_add	Tests function <code>vec_add()</code> . Add test vectors and compares results with precalculated data.
test_vec_bexp	Tests function <code>vec_bexp()</code> . Checks on test vectors of different sizes comparing with direct computation using <code>S_exp0_1()</code> function.
test_vec_shift	Tests function <code>vec_shift()</code> . Checks on test vectors with different shift counts.
test_vec_recip16	Tests function <code>vec_recip16()</code> . Checks on test vector by multiplication of reciprocal to original denominator.
test_vec_divide	Tests function <code>vec_divide()</code> . Checks on test vectors by multiplication of result to original denominator.
test_vec_log	Tests functions <code>vec_log()</code> , <code>vec_log2()</code> , <code>vec_log10()</code> . Checks on test vectors by comparing with precalculated data.
test_vec_sine	Tests functions <code>vec_sine()</code> , <code>vec_cosine()</code> . Checks on test vectors by comparing with precalculated data.
test_vec_tan	Tests function <code>vec_tan()</code> . Checks on test vectors by comparing with precalculated data.
test_vec_atan2	Tests function <code>vec_atan2()</code> . Checks on test vectors by comparing with precalculated data.
test_vec_atan	Tests function <code>vec_atan()</code> . Checks on test vectors by comparing with precalculated data.

3.5 IIR tests

Subroutine	Test procedure
test_iir_latr	Tests function <code>iir_latr()</code> . Passes the signal read from file by blocks of 80 samples through the lattice filter of 6 sections and write filtered output to the file.
test_iir_latc	Tests function <code>iir_latc()</code> . Passes the signal read from file by blocks of 80 samples through the lattice filter of 6 sections and write filtered output to the file.
test_iir_latd	Tests function <code>iir_latd()</code> . Passes the signal read from file by blocks of 80 samples through the lattice filter of 6 sections and write filtered output to the file.
test_iir_bqc	Tests function <code>iir_bqc()</code> . Passes the signal read from file by blocks of 80 samples through the 6-th section buquad filter and write filtered output to the file.
test_iir_bqd	Tests function <code>iir_bqd()</code> . Passes the stereo signal read from file by blocks of 80 samples buquad filters and write filtered output to the file. Two different filters are used: for left channel it uses filter with 6 sections, for right – single section peak filter.

3.6 FFT tests

Subroutine	Test procedure
test_fft_cplx	Tests functions <code>fft_cplx()</code> , <code>ifft_cplx()</code> . Passes the signal read from file by blocks through the FFT filter and write filtered output to the file.
test_fft_real	Tests functions <code>fft_real()</code> and <code>ifft_real()</code> . Passes the signal read from file by blocks through the FFT filter and write filtered output to the file.

Chapter 4 Customer support

If you have questions, want to report problems or suggestions regarding the **NatureDSP Signal** library or want to port this library to another platforms, contact **IntegrIT** Ltd. at support@integrit.ru.