NatureDSP Signal Library for HiFi 3/HiFi 3z DSP

Digital Signal Processing

Library Reference

Library Release 5.0.0
API Revision 4.00
Mar, 2025

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
https://support.cadence.com/

IMPORTANT NOTICE

Table of Contents

Document History 6
Preface 8
About This Manual 8
Supported Targets 8
About GitHub and XPG release 8
About this Release 8
Notations 9
Abbreviations 9
1 General Library Organization 10
1.1 Headers 10
1.2 Static Variables and Usage of C Standard Libraries 10
1.3 Types 10
1.4 Fractional Formats 11
1.5 Compiler Requirements 11
1.6 Call Conventions 11
1.7 Overflow Control and Intermediate Data Format 12
1.8 Exceptions and Processor Control Registers 12
1.9 Special Numbers 12
1.10 Endianess 12
1.11 Performance Issues 13
1.12 Object Model 13
1.13 Brief Function List 13
2 Reference 16
2.1 FIR Filters and Related Functions 16
2.1.1 Block Real FIR Filter 16
2.1.2 Block Real FIR Filter with Arbitrary Parameters 18
2.1.3 Complex Block FIR Filter 20
2.1.4 Decimating Block Real FIR Filter 22
2.1.5 Interpolating Block Real FIR Filter 24
2.1.6 Circular convolution 26
2.1.7 Linear Convolution 28
2.1.8 Circular Correlation 29
2.1.9 Linear Correlation 30
2.1.10 Circular Autocorrelation 31
2.1.11 Linear Autocorrelation 32
2.1.12 Blockwise Adaptive LMS Algorithm for Real Data 33
2.2 IIR filters 35
2.2.1 Bi-quad Real Block IIR 35

2.2.2 Lattice Block Real IIR 39

2.3 Mathematics 41
2.3.1 Reciprocal on Q31/Q15 Numbers 42
2.3.2 Division of Q31/Q15 Numbers 43
2.3.3 Logarithm 44
2.3.4 Antilogarithm 45
2.3.5 Square Root 47
2.3.6 Reciprocal Square Root 48
2.3.7 Sine/Cosine 48
2.3.8 Tangent 50
2.3.9 Arctangent 51
2.3.10 Full Quadrant Arctangent 52
2.3.11 Hyperbolic Tangent 53
2.3.12 Sigmoid 53
2.3.13 Softmax 54
2.3.14 Integer to Float Conversion 54
2.3.15 Float to Integer Conversion 55

2.4 Complex Mathematics 56
2.4.1 Complex Magnitude 56
2.4.2 Complex Inverse Magnitude 56
2.4.3 Complex to Complex Multiplication 57
2.4.4 Complex Vector to Real Vector Multiplication 57
2.4.5 Complex Vector to Real Scalar Multiplication 58
2.4.6 Complex Conjugate 59

2.5 Vector Operations 60
2.5.1 Vector Dot Product 60
2.5.2 Vector Sum 61
2.5.3 Power of a Vector 61
2.5.4 Vector Scaling with Saturation 62
2.5.5 Common Exponent 64
2.5.6 Elementwise Absolute of Vector 66
2.5.7 Vector Min/Max 66
2.5.8 Elementwise Vector Subtraction 67
2.5.9 Elementwise Vector Multiplication 68
2.5.10 Vector Sum 68
2.5.11 Vector Mean 69
2.5.12 Vector RMS 69
2.5.13 Vector Variance 70
2.5.14 Vector Standard Deviation 70

2.6 Matrix Operations 72
2.6.1 Matrix Multiply 72
2.6.2 Matrix by Vector Multiply 74

2.7 Matrix Decomposition/Inversion 76
2.7.1 Matrix Inverse 76

2.8 Fitting/Interpolation 77
2.8.1 Polynomial Approximation 77

2.9 Fast Fourier Transforms 78
2.9.1 FFT on Complex Data 80

2.9.2 FFT on Real Data 81
2.9.3 Inverse FFT on Complex Data 83
2.9.4 Inverse FFT Forming Real Data 84
2.9.5 FFT on Complex Data with Optimized Memory Usage 85
2.9.6 FFT on Real Data with Optimized Memory Usage 86
2.9.7 Inverse FFT on Complex Data with Optimized Memory Usage 88
2.9.8 Inverse FFT on Real Data with Optimized Memory Usage 89
2.9.9 Discrete Cosine Transform 90
2.9.10 Modified Discrete Cosine Transform 91
2.9.11 2D Discrete Cosine Transform 92
2.9.12 2D Inverse Discrete Cosine Transform 93

2.10 Identification Routines 95
2.10.1 Library Version Request 95
2.10.2 Library API Version Request 95
2.10.3 Library API Capability Request 95

3 Test Environment and Examples 96

3.1 Supported Use Environment, Configurations and Targets 96

3.2 Importing the Workspaces in Xtensa Xplorer 96
3.2.1 Build and Run NatureDSP Library under XtensaXplorer IDE 96
3.2.2 Build and Run the NatureDSP Signal Library with Xtensa Compiler using Command-Line Tools 99
3.2.3 Command-line Options 100

4 Appendix 102

4.1 Matlab Code for Conversion of SOS Matrix to Coefficients of IIR Functions 102
4.1.1 bqriir24x24_df1 conversion 102
4.1.2 bqriir16x16_df1, bqriir32x16_df1 conversion 103
4.1.3 bqriir24x24_df2 conversion 104
4.1.4 bqriir16x16_df2, bqriir32x16_df2 conversion 105
4.1.5 bqriir32x32_df1 conversion 106
4.1.6 bqriir32x32_df2 conversion 107
4.1.7 bqriirf_df1, bqriirf_df2, bqriirf_df2t conversion 108

4.2 Matlab Code for Generation the Twiddle Tables 108
4.2.1 Twiddles for fft_cplx24x24_ie, ifft_cplx24x24_ie, fft_real24x24_ie, ifft_real24x24_ie 109
4.2.2 Twiddles for fft_cplx32x16_ie, ifft_cplx32x16_ie, fft_real32x16_ie, ifft_real32x16_ie, fft_cplx16x16_ie, ifft_cplx16x16_ie, fft_real16x16_ie, ifft_real16x16_ie 109
4.2.3 Twiddles for fft_cplx32x32_ie, ifft_cplx32x32_ie, fft_real32x32_ie, ifft_real32x32_ie 109
4.2.4 Twiddles for fft_cplxf_ie, ifft_cplxf_ie, fft_realf_ie, ifft_realf_ie 109

Document History

Revision	Date	Major changes
1.44	April, 2012	Initial version
3.1	September, 2015	- bug fixes (wrong operation on negative vector size) - added floating data types and VFPU support - changed alignment requirements to complex data types - improved accuracy for reciprocal, division, antilogarithm - added Matlab code (Appendix 4) for generation of filter coefficients and FFT twiddle tables
3.11	November, 2015	- typo corrections - added Library API Capability Request
3.12	January, 2016	- changed behavior of sine/cosine/tangent – now they do not generate invalid exception when argument is too big - code is adopted to RF-2015.3 tools
3.13	February, 2016	- fixed mistake in the description of antilogarithm functions
3.14	November, 2016	- added 24-bit Full-quadrant arc tangent (vec_atan2_24x24(), scl_atan2_24x24()) - improved Matlab code for IIR coefficient conversion
3.15	March, 2017	- added verification reporting `(-vreport` command line option)
3.20	August, 2017	- added HiFi3z capabilities - new packaging, added more flexible options for testing - new FIR functions: bkfir16x16_xxx, bkfira16x16_xxx, bkfira32x32_xxx, cxfir16x16_xxx, bkfir32x32_xxx, firdec16x16_xxx, firdec32x32_xxx, firinterp16x16_xxx, firinterp32x32_xxx, fir_convol16x16, fir_convola16x16, fir_convol32x32, fir_convola32x32, fir_xcorr16x16, fir_xcorra16x16, fir_xcorr32x32, fir_xcorra32x32, fir_acorr16x16, fir_acorra16x16, fir_acorr32x32, fir_acorra32x32, lconvola16x16, lconvola32x32, lxcorra16x16, lxcorra32x32, lacorra16x16, lacorra32x32, fir_blms16x16, fir_blms32x32 - new IIR functions: bqriir16x16_df1_xxx, bqriir16x16_df2_xxx, bqriir32x32_df1_xxx, latr16x16_xxx, latr32x32_xxx - new vector functions: vec_dot32x32, vec_dot32x32_fast, vec_scale32x32, vec_scale32x32_fast - new math functions: vec_sqrt16x16, vec_sqrt64x32, scl_sqrt16x16, scl_sqrt64x32, vec_rsqrt16x16, vec_rsqrt32x32, scl_rsqrt16x16, scl_rsqrt32x32, vec_tanh32x32, scl_tanh32x32, vec_sigmoid32x32, scl_sigmoid32x32, vec_softmax32x32 - added matrix/matrix and matrix/vector multiplies for 32x32 - fixed point matrix/matrix APIs is changed (inputs and outputs are represented in usual matrix layout, not via pointers to rows) - FFT: added 32x32 FFT (fft_cplx32x32, ifft_cplx32x32, fft_real32x32, ifft_real32x32, fft_cplx32x32_ie, ifft_cplx32x32_ie, fft_real32x32_ie, ifft_real32x32_ie), 16x16 FFT with dynamic scaling (fft_cplx16x16_ie, ifft_cplx16x16_ie, fft_real16x16_ie, ifft_real16x16_ie), mixed radix 32x32 FFTs - changed parameter N of DCT-II function to DCT-II handle to minimize memory footprint - DCT: added DCT type II, MDCT, IMDCT and 2D DCT
3.21	September, 2017	- 2D-DCT definition made compatible with ITU-T.81 (JPEG compression) - added inverse 2D-DCT
3.22	November, 2017	- added size N=64 for DCT Type II - added more sizes for mixed radix FFT/IFFT (complex: 80, 100, 160, 200, 384, 400, 600, real: 30, 90, 384, 720, 1152, 1440, 1536, 1920) - added separate tests for real mixed radix FFTs (-rnfft) and for fast vectorized math (-mathvf) - more explanations about FFT scaling modes

Revision	Date	Major changes
3.30	January, 2018	- merged with HiFi4 API - added extended IR for bkfir, cxfir - added FIR functions with 72-bit extended accumulation (32x32ep) - for HiFi4 only - 64-bit vector dot products (for HiFi3/3z) - 64x32 integer division (for HiFi3/3z) - added `-func`, `-brief` options to the test harness framework - added support of Xtensa Xplorer workspaces
4.00	March 2025	- Added 43 new kernels under complex and vector (float, int32 and int16 versions) - Updated the testdriver - Fixed RJ3 and RJ4 related warnings and errors - Fixed RJ3 functional failures
4.10	April 2025	- Updated the document for XWS related build process
4.11	May 2025	- Updated the document as per AE review comments for release V5_0_0

Preface

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. Library supports both fixed-point and single precision floating data types.

Supported Targets

Library supports Cadence HiFi3/HiFi3z with VFPU little endian targets.
In general, library API is the same for all supported cores, but some functionality might be missing depending on core abilities. Common rules are:

- functions with floating point inputs/outputs require VFPU/SFPU option of the core

Presence of specific functions might be detected in runtime using library identification routines, see para 2.10.3.

About GitHub and XPG release

NatureDSP library release packages are available in two locations:

1. General Availability (GA) release in XPG repository: The GA release package includes major updates, released after comprehensive tests and stability checks. XPG releases are scheduled at longer intervals.
2. Incremental revision in GitHub repository: Stable version includes incremental fixes/enhancements on top of XPG/GA release. The revised source codes in this version go through feature specific tests, regression tests and stability tests. GitHub releases can be more frequent, as the intension is to make the stable version available quickly for the users. However, compared to XPG releases, GitHub releases are tested on the limited set of processor configurations. Also, the reference manual and performance benchmarks might not be updated in these releases.

   GitHub repository location: https://github.com/foss-xtensa/ndsplib-hifi3z

About this Release

This is version 5.0.0 of the HiFi 3/3z NDSP library which is tested on the Xtensa Xplorer (11.1.4) and xtensa tools version RJ-2024.4. This release additionally contains optimizations supported by the xt-clang C/C++ compiler. Please refer to release notes for details. This library has functions fine-tuned for better performance for HiFi 3/3z DSP cores on RJ-2024.4 Xtensa SW tools, and floating-point variants require HiFi 3/3z DSP with SP-VFPU option enabled.

Owing to many optional features & add-ons available with HiFi 3/3z core, many different variants of HiFi 3/3z configuration can be created by the user. The present HiFi 3/3z NatureDSP Library is tested on the HiFi 3/3z configurations that use (i) Xtensa C library, (ii) 2 Data RAM banks and (iii) big on-chip memory using RJ-2024.4 tools.

Benchmark performance data is published in a separate document.

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

API	Application program interface
DCT	Discrete Cosine Transform
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

1 General Library Organization

1.1 Headers

The library is delivered in several packages. The API for each package is defined in the appropriate header file which describes particular functions in the package. When the appropriate `#include` preprocessor directive is included in your source, the compiler uses the prototypes to check that each function is called with the correct arguments.

`./library/include/NatureDSP_types.h`	Declarations of basic data types and compiler auto detection	1.3
`./library/include/NatureDSP_Signal.h`	Declarations of all library functions	
`./library/include/NatureDSP_Signal_fir.h`	FIR Filters and Related Functions	2.1
`./library/include/NatureDSP_Signal_iir.h`	IIR Filters	2.2
`./library/include/NatureDSP_Signal_math.h`	Math Functions	2.3
`./library/include/NatureDSP_Signal_complex.h`	Complex Math Functions	2.4
`./library/include/NatureDSP_Signal_vector.h`	Vector Operations	2.4.2
`./library/include/NatureDSP_Signal_matop.h`	Matrix Operations	2.5.8
`./library/include/NatureDSP_Signal_matinv.h`	Matrix Decomposition and Inversion Functions	2.7
`./library/include/NatureDSP_Signal_fit.h`	Fitting/interpolation	2.8
`./library/include/NatureDSP_Signal_fft.h`	FFT/DCT Routines	2.9
`./library/include/NatureDSP_Signal_id.h`	Identification functions	2.10
`./library/include/NatureDSP_Signal_diag.h`	internal APIs for diagnostics	

1.2 Static Variables and Usage of C Standard Libraries

All library functions are re-entrant. Library functions do not call functions from standard C-library.

1.3 Types

Library uses the following C types with defined length

Name	Description	Alignment, bytes
`f24`	24-bit fractional type	4
`int16_t`	16-bit signed value	2
`int32_t`	32-bit signed value	4
`uint32_t`	32-bit unsigned value	4
`int64_t`	64-bit signed value	8
`float32_t`	32-bit single precision floating point value	4
`complex_float`	complex single precision floating point (pair of two 32-bit values)	8
`complex_fract16`	complex 16-bit factional value (pair of two 16-bit values)	4
`complex_fract32`	complex 32-bit factional value (pair of two 32-bit values)	8

It is assumed throughout this Reference that constant pointers passed through function arguments point at read-only data

Normally, `f24` fractional data are stored 3 higher bytes of 32-bit words and 8 LSBs are ignored, however, few routines use packed 24-bit data where 24-bit fractional numbers allocates only 3 consecutive bytes.
Data of given type should be aligned on its `sizeof()`, see table above.

1.4 Fractional Formats

Natively, HiFi3/HiFi3z CPU uses special fractional type `f24` which is stored in a memory as 32 bit word keeping significant bits in bits 8 through 31. So, from that perspective it may be treated as `Q31` number. But users should take into account that 8 LSB are ignored. **Unless specifically noted, library functions use that `Q31` format, or, in another words, `Q0.31`**.
In a `Qm.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 `Qm.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 `Qn`.

Example data type and their formats are collected in the table below:

Data type	Format	Range	Resolution	Minimum value	Maximum value
int16_t	Q0.15	-1 … 0,999969	3e-5	-32768	32767
int16_t	Q6.9	-64 … 63,998	2e-3	-32768	32767
int32_t	Q1.30	-2 … 1,9999999991	9e-10	-2147483648	2147483647
int32_t	Q0.31	-1 … 0,9999999995	5e-10	-2147483648	2147483647
int32_t	Q6.25	-64… 63,999999970	3e-8	-2147483648	2147483647
int32_t	Q16.15	-65536… 65535,99997	3e-5	-2147483648	2147483647
f24	Q1.30	-2 … 1,9999997625	2e-7	-2147483648	2147483392
f24	Q0.31	-1 … 0,9999998784	1e-7	-2147483648	2147483392
f24	Q6.25	-64… 63,99999240	8e-6	-2147483648	2147483392
f24	Q16.15	-65536…65535,9921875	8e-3	-2147483648	2147483392

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

When building the library source files or library-dependent modules it is assumed that the target is a Cadence processor implementing the Xtensa HiFi3/HiFi3z Audio Engine Instruction Set Architecture with VFPU option.

1.6 Call Conventions

Library uses ANSI-C call conventions.

1.7 Overflow Control and Intermediate Data Format

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. However, if possible library use saturated arithmetic to prevent overflows.
In the most fixed-point routines operating with summing of multiple elements (i.e. FIR, matrix multiplies, etc.), library stores intermediate values in 64-bit accumulators using Q16.47 fixed-point representation thus protecting from the overflows in the intermediate stages. Floating point routines use single precision floating point format for storing intermediate data.
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 Exceptions and Processor Control Registers

Except for some mathematical routines, compatible with IEEE-754 and C99 standards (see para 2.3), all library functions do not touch global `errno` variable and do not modify the FPU enabled bits. FPU flags may be set during the execution of the routines. It is up to the caller to decide how to proceed given the flags.

Example of use cases are:

- The caller could enable floating point control bits before calling functions. This would result in an external signal that indicates an exceptional condition has occurred. We expect the customer to use that signal to control an external interrupt – thus enabling an imprecise interrupt.
- The caller could zero the status flags before a function and check them when the function returns to see if any exceptional conditions occurred.

1.9 Special Numbers

The IEEE754 standard specifies some special values, and their representation: positive infinity ($+\infty$ or `+Inf`), negative infinity ($-\infty$ or `-Inf`), a negative zero (`-0`) distinct from ordinary ("positive") zero (`+0`), and "not a number" values (`NaN`s). In general, the following rules are applied:

- negative zero is treated as usual negative number
- the result of operations under `NaN` is `NaN`
- operations with infinity return `NaN` except for few routines which require to interpret only the sign of infinity
- If a result depends on several values (E.g. in filters and correlations), and one or more of them is `NaN` or `Inf`, the propagation of those special values is complicated. The library routines will propagate the value in a way that minimizes cycles and code size. A special value will still appear in the output.
- outputs for mathematical functions for special numbers on their inputs follows ISO/IEC 9899 if not explicitly mentioned

1.10 Endianess

Library supports little-endian mode.

1.11 Performance Issues

Real-time performance of all functions depends on fulfillment special restrictions applied to input/output arguments. Typically, for maximum performance, user have to use **aligned data arrays (on 8 byte boundary)** for storing input and output arguments, number of data should be **multiple of 2 or 4** and should be **greater than 4**.
Specific requirements are given for each function in its API description, however, for most of the kernels the output alignment requirement is same as input alignment requirements and output alignment requirement is not mentioned in the API description.
Data alignment may be achieved by several methods:

- placing the data into special data section and make alignment at the link-time
- use `__attribute__((aligned(x)))` modifiers in the data declarations
- dynamically allocate arrays of slighter bigger size and align pointers[1]

Test examples use two last methods.

1.12 Object Model

Effective use of all HiFi3/HiFi3z core benefits require specific processing and special data moves minimizing the overhead. That is why many functions are supplied with object-like interface simplifying real-time processing chain but requiring special initialization before processing. Besides, function wrapped by object-like interface use best possible alignment for data storage and may utilize HiFi3/HiFi3z core better in some cases.
Initialization normally done once at the initialization time and do not affect to the real-time performance. Sequence consists of three stages

- call `<obj>_alloc()` function with parameters that define the block size, filter length, etc. This function/macro returns the size of memory has to be allocated for object for those specific parameters
- allocate the memory somehow. It may be done dynamically if `<obj>_alloc()` function is used
- pass the pointer to allocated memory to the function `<obj>_init`. It cleans up that memory block, reorder filter coefficients appropriately, etc. and returns the handle to the object. This handle will be used later for data processing by this given object, .i.e., block filtering.

Here we denote the symbolic name of object as `<obj>`. For example, corresponding functions for block FIR filtering will be named as:

`bkfir_alloc()`	request the memory size for object
`bkfir_init()`	initialize the object
`bkfir_process()`	make filtering of block

1.13 Brief Function List

Vectorized version	Scalar version	Purpose	Reference
FIR filters and related functions			
bkfir		Block real FIR filter	2.1.1, 2.1.2
cxfir		Complex block FIR filter	2.1.3
firdec		Decimating block real FIR filter	2.1.4

[1] Xtensa C/C++ compiler's `malloc()` always returns pointer aligned on 64-bit boundary special additional alignment procedure is not required

Vectorized version	Scalar version	Purpose	Reference
`firinterp`		Interpolating block real FIR filter	2.1.5
`fir_convol`, `cxfir_convol`		Circular/linear convolution	2.1.6, 2.1.7
`fir_xcorr`		Circular/linear correlation	2.1.8, 2.1.9
`fir_acorr`		Circular/linear autocorrelation	2.1.10, 2.1.11
`fir_blms`		Blockwise Adaptive LMS algorithm	2.1.12
IIR filters			
`bqriir, bqciir`		Biquad Real block IIR	2.2.1
`latr`		Lattice block Real IIR	2.2.2
Mathematics			
`vec_recip`	`scl_recip`	Reciprocal on a vector of Q31 numbers	2.3.1
`vec_divide`	`scl_divide`	Division	2.3.2

`vec_logn`
`vec_log2`
`vec_logn`
	`scl_logn`
	`scl_log2`
	`scl_logn`
		Different kinds of logarithm	2.3.3

`vec_recip`	`scl_recip`	Reciprocal on a vector of Q31 numbers	2.3.1
`vec_divide`	`scl_divide`	Division	2.3.2

`vec_logn`
`vec_log2`
`vec_logn`
	`scl_logn`
	`scl_log2`
	`scl_logn`
		Different kinds of logarithm	2.3.3

`vec_antilog2`
`vec_antilog10`
`vec_antilogn`
	`scl_antilog2`
	`scl_antilog10`
	`scl_antilogn`
		Different kinds of antilogarithm	2.3.4

`vec_sqrt`	`scl_sqrt`	Square root	2.3.5
`vec_rsqrt`	`scl_rsqrt`	Reciprocal square root	2.3.6
`vec_sine`	`scl_sine`	Sine	2.3.7
`vec_cosine`	`scl_cosine`	Cosine	
`vec_tan`	`scl_tan`	Tangent	2.3.8
`vec_atan`, `vec_atan2`	`scl_atan, scl_atan2`	Arctangent	2.3.9, 2.3.10
`vec_tanh32x32`	`vec_tanh32x32`	Hyperbolic tangent	2.3.11
`vec_sigmoid32x32`	`scl_sigmoid32x32`	Sigmoid	2.3.12
`vec_softmax32x32`		Softmax	2.3.13
`vec_int2float`	`scl_int2float`	Integer to float conversion	2.3.14
`vec_float2int`	`scl_float2int`	Float to integer conversion	2.3.15
Complex Mathematics			
`vec_complex2mag`	`scl_complex2mag`	Complex magnitude	2.4.1
`vec_complex2invmag`	`scl_complex2invmag`	Reciprocal of complex magnitude	2.4.1
Vector operations			
`vec_dot`		Vector dot product	2.5.1
`vec_add`		Vector sum	2.5.2
`vec_power`		Power of a vector	2.5.3
`vec_shift` `vec_scale`		Vector scaling with saturation	2.5.4
`vec_bexp`	`scl_bexp`	Common exponent	2.5.5
`vec_min, vec_max`		Find a maximum/minimum in a vector	2.5.6
Matrix operations			
`mtx_mpy`		Matrix multiply	2.6.1
`mtx_vecmpy`		Matrix by vector multiple	2.6.2
Matrix Decomposition and Inversion Functions			
`mtx_inv`		Matrix inversion	2.7.1

Vectorized version	Scalar version	Purpose	Reference
Fitting/Interpolation			
`vec_poly`		Polynomial approximation	2.8.1
FFT/DCT			
`fft_cplx`		FFT on complex data	2.9.1
`fft_real`		FFT on real data	2.9.2
`ifft_cplx`		Inverse FFT on complex data	2.9.3
`ifft_real`		Inverse FFT forming real data	2.9.4
`fft_cplx_ie`		FFT on complex data with optimized memory usage	2.9.5
`fft_real_ie`		FFT on real data with optimized memory usage	2.9.6
`ifft_cplx_ie`		Inverse FFT on complex data with optimized memory usage	2.9.7
`ifft_real_ie`		Inverse FFT forming real data with optimized memory usage	2.9.8
`dct,mdct`		Discrete cosine transform	2.9.9, 2.9.10
`dct2d, idct2d`		2D Discrete cosine transforms	2.9.11, 2.9.12
Identification			
`NatureDSP_Signal_get_library_version`		Library Version Request	2.10.1
`NatureDSP_Signal_get_library_api_version`		Library API Version Request	2.10.2
`NatureDSP_Signal_isPresent`		Library API Capability Request	2.10.3

2 Reference

2.1 FIR Filters and Related Functions

FIR filtering APIs excepting correlation/convolution, autocorrelation and blockwise LMS algorithm require instantiation. In particular, filter objects encapsulate the delay line buffer, which is organized in such a way that advanced processor capabilities (e.g. circular data addressing) are efficiently utilized. When allocating and initializing a filter instance through `xfir_alloc()` and `xfir_init()` function calls, the user has to specify the length of filters and its coefficients. On the data processing stage the user application sequentially calls an `xfir_process()` function, providing it with a block of `N` input samples on each call. `xfir_process()` function updates the internal delay line with input samples, and computes `N` filter output samples, which are returned to the calling application via the output data buffer argument.

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 `y`. The filter calculates `N` output samples using `M` coefficients and requires last `M-1` samples in the delay line which is updated in circular manner for each new sample.
User has an option to set IR externally or copy from original location (i.e. from the slower constant memory). In the first case, user is responsible for right alignment, ordering and zero padding of filter coefficients – usually array is composed from zeroes (left padding), reverted IR and right zero padding.

Precision

7 variants available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit outputs
24x24	24-bit data, 24-bit coefficients, 24-bit outputs. Available for HiFi3/HiFi3z cores only
24x24p	use 24-bit data packing for internal delay line buffer and internal coefficients storage
32x16	32-bit data, 16-bit coefficients, 32-bit outputs
32x32	32-bit data, 32-bit coefficients, 32-bit outputs
32x32ep	32-bit data, 32-bit coefficients, 32-bit outputs, use 72-bit accumulators for intermediate computations. Available for HiFi4 only
f	floating point. Requires VFPU core option

Algorithm

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

NOTE:
This is formal description of algorithm, in reality processing in done using circular buffers, so user application is not responsible for management of delay lines

Object allocation

```
size_t bkfir16x16_alloc (int M, int extIR)
size_t bkfir24x24_alloc (int M, int extIR)
size_t bkfir24x24p_alloc(int M, int extIR)
size_t bkfir32x16_alloc (int M, int extIR)
size_t bkfir32x32_alloc (int M, int extIR)
size_t bkfir32x32ep_alloc(int M, int extIR)
size_t bkfirf_alloc (int M, int extIR)
```

Type	Name	Size	Description
Input			
`int`	`M`		length of filter, should be a multiple of 4
`int`	`extIR`		if zero, IR is copied from original location, otherwise not but user should keep alignment, order of

			coefficients and zero padding requirements shown below

Returns: size of memory in bytes to be allocated

NOTE:
Approximate amount of requested memory is listed below

Function	Approximate memory requirements, bytes	
	extIR=0	**extIR!=0**
`bkfir16x16_alloc`	72+M*4	56+M*2
`bkfir24x24_alloc`	72+M*8	48+M*4
`bkfir24x24p_alloc`	80+M*6	56+M*3
`bkfir32x16_alloc`	72+M*6	64+M*4
`bkfir32x32_alloc`	72+M*8	48+M*4
`bkfir32x32ep_alloc`	72+M*8	48+M*4
`bkfirf_alloc`	72+M*8	48+M*4

Object initialization

```
Bkfir16x16_handle_t bkfir16x16_init
           (void * objmem, int M, int extIR, const int16_t *  h)
bkfir24x24_handle_t bkfir24x24_init
           (void * objmem, int M, int extIR, const f24 *  h)
bkfir24x24p_handle_t bkfir24x24p_init
           (void * objmem, int M, int extIR, const f24 *  h)
bkfir32x16_handle_t bkfir32x16_init
           (void * objmem, int M, int extIR, const int16_t*  h)
bkfir32x32_handle_t bkfir32x32_init
           (void * objmem, int M, int extIR, const int32_t*  h)
bkfir32x32ep_handle_t bkfir32x32ep_init
           (void * objmem, int M, int extIR, const int32_t* restrict h)
bkfirf_handle_t bkfirf_init
           (void * objmem, int M, int extIR, const float32_t*  h)
```

Type	Name	Size	Description
Input			
`void*`	`objmem`		allocated memory block
`f24,int16_t, int32_t, float32_t`	`h`	`M`	filter coefficients; `h[0]` is to be multiplied with the newest sample
`int`	`M`		length of filter
`int`	`extIR`		if zero, IR is copied from original location, otherwise not but user should keep alignment, order of coefficients and zero padding requirements shown below

Returns: handle to the object

Alignment, ordering and zero padding for external IR (`extIR!=0`)

Function	Alignment, bytes	Left zero padding, bytes	Coefficient order	Right zero padding, bytes
`bkfir16x16_init`	8	2	inverted	6
`bkfir24x24_init`	8	4	inverted	12
`bkfir24x24p_init`	8	((-M&4)+5)*3	inverted	7
`bkfir32x16_init (M>32)`	8	10	inverted	6
`bkfir32x16_init (M<=32)`	8	2	inverted	6
`bkfir32x32_init`	8	4	inverted	12
`bkfir32x32ep_init`	8	4	inverted	12
`bkfirf_init`	8	0	direct	0

Update the delay line and compute filter output

```
void bkfir16x16_process (
              bkfir16x16_handle_t  handle,
              int16_t *  y, const int16_t *  x, int N )
void bkfir24x24_process (
              bkfir24x24_handle_t  handle,
              f24 *  y, const f24 *  x, int N )
void bkfir24x24p_process(
              bkfir24x24p_handle_t handle,
              f24*  y,  const f24 *  x, int N )
void bkfir32x16_process (
              bkfir32x16_handle_t  handle,
              int32_t *  y, const int32_t *  y, int N)
void bkfir32x32_process (
              bkfir32x32_handle_t  handle,
              int32_t *  y, const int32_t *  y, int N)
void bkfir32x32ep_process ( bkfir32x32ep_handle_t  handle,
                      int32_t * y, const int32_t * y, int N)
void bkfirf_process (
              bkfirf_handle_t  handle,
              float32_t * y, const float32_t * x, int N);
```

Type	Name	Size	Description
Input			
`int16_t, f24, int32_t, float32_t`	`x`	`N`	input samples
`int`	`N`		length of sample block
Output			
`int16_t, f24, int32_t, float32_t`	`y`	`N`	output samples

Returns: none

Restrictions

`x,y` – should not overlap
`x,h` - aligned on a 8-bytes boundary
`N,M` - multiples of 4

2.1.2 Block Real FIR Filter with Arbitrary Parameters

Description

These functions implement FIR filter described in previous chapter with no limitation on size of data block, alignment and length of impulse response for the cost of performance.

Precision

6 variants available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit outputs
24x24	24-bit data, 24-bit coefficients, 24-bit outputs. Available for HiFi3/HiFi3z cores only
32x16	32-bit data, 16-bit coefficients, 32-bit outputs
32x32	32-bit data, 32-bit coefficients, 32-bit outputs
32x32ep	32-bit data, 32-bit coefficients, 32-bit outputs, use 72-bit accumulators for intermediate computations. Available for HiFi4 only
f	floating point. Requires VFPU core option

Algorithm

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

NOTE:
This is formal description of algorithm, in reality processing in done using circular buffers, so user application is not responsible for management of delay lines

Object allocation

```
size_t bkfira16x16_alloc(int M)
size_t bkfira24x24_alloc(int M)
size_t bkfira32x16_alloc(int M)
```

```
size_t bkfira32x32_alloc(int M)
size_t bkfira32x32ep_alloc(int M)
size_t bkfiraf_alloc(int M)
```

Type	Name	Size	Description
Input			
int	M		length of filter

Returns: size of memory in bytes to be allocated

Approximate amount of requested memory is listed below

Function	Approximate memory requirements, bytes
bkfira16x16_alloc	72+ M*4
bkfira32x16_alloc	72+ M*6
bkfira24x24_alloc	80+ M*8
bkfira32x32_alloc	80+ M*8
bkfira32x32ep_alloc	80+ M*8
bkfiraf_alloc	80+ M*8

Object initialization

```
bkfira16x16_handle_t bkfira16x16_init
           (void * objmem, int M, const int16_t *  h)
bkfira24x24_handle_t bkfira24x24_init
           (void * objmem, int M, const f24 *  h)
bkfira32x16_handle_t bkfira32x16_init
           (void * objmem, int M, const int16_t*  h)
bkfira32x32_handle_t bkfira32x32_init
           (void * objmem, int M, const int32_t*  h)
bkfira32x32ep_handle_t bkfira32x32ep_init
           (void * objmem, int M, const int32_t*  h)
bkfiraf_handle_t bkfiraf_init
           (void * objmem, int M, const int16_t*  h)
```

Type	Name	Size	Description
Input			
void*	objmem		allocated memory block
f24,int16_t, int32_t, float32_t	h	M	filter coefficients; `h[0]` is to be multiplied with the newest sample
int	M		length of filter

Returns: handle to the object

Update the delay line and compute filter output

```
void bkfira16x16_process (
                  bkfira16x16_handle_t  handle,
                  int16_t *  y, const int16_t *  x, int N );
void bkfira24x24_process (
                  bkfira24x24_handle_t  handle,
                  f24 *  y, const f24 *  x, int N );
void bkfira32x16_process (
                  bkfira32x16_handle_t  handle,
                  int32_t *  y, const int32_t *  y, int N);
void bkfira32x32_process (
                  bkfira32x32_handle_t  handle,
                  int32_t *  y, const int32_t *  y, int N);
void bkfira32x32ep_process (
                  bkfira32x32ep_handle_t  handle,
                  int32_t *  y, const int32_t *  y, int N);
void bkfiraf_process (
               bkfiraf_handle_t  handle,
               float32_t * y, const float32_t * x, int N);
```

Type	Name	Size	Description
Input			
int16_t, f24, int32_t, float32_t	x	N	input samples
int	N		length of sample block
Output			
int16_t, f24, int32_t, float32_t	y	N	output samples

Returns: none

Restrictions

x,y – should not overlap

2.1.3 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 y. The filter calculates N output samples using M coefficients, requires last M-1 samples in the delay line which is updated in circular manner for each new sample. Real and imaginary parts are interleaved and real parts go first (at even indexes).
User has an option to set IR externally or copy from original location (i.e. from the slower constant memory). In the first case, user is responsible for right alignment, ordering and zero padding of filter coefficients – usually array is composed from zeroes (left padding), reverted IR and right zero padding.

Precision

6 variants available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit outputs
24x24	24-bit data, 24-bit coefficients, 24-bit outputs. Available for HiFi3/HiFi3z cores only
32x16	32-bit data, 16-bit coefficients, 32-bit outputs
32x32	32-bit data, 32-bit coefficients, 32-bit outputs
32x32ep	32-bit data, 32-bit coefficients, 32-bit outputs, use 72-bit accumulators for intermediate computations. Available for HiFi4 only
f	floating point. Requires VFPU core option

Algorithm

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

NOTE:
This is formal description of algorithm, in reality processing in done using circular buffers, so user application is not responsible for management of delay lines

Object allocation

```
size_t cxfir16x16_alloc(int M, int extIR)
size_t cxfir24x24_alloc(int M, int extIR)
size_t cxfir32x16_alloc(int M, int extIR)
```

```
size_t cxfir32x32_alloc(int M, int extIR)
size_t cxfir32x32ep_alloc(int M, int extIR)
size_t cxfirf_alloc(int M, int extIR)
```

Type	Name	Size	Description
Input			
int	M		length of filter

Returns: size of memory in bytes to be allocated
NOTE:
Approximate amount of requested memory is listed below

Function	Approximate memory requirements, bytes	
	extIR=0	extIR!=0
cxfir16x16_alloc, HiFi3	80+12*M	64+8*M
cxfir16x16_alloc, HiFi3z/4	80+12*M	40+4*M
cxfir32x16_alloc	80+12*M	64+8*M
cxfir24x24_alloc	64+16*M	72+8*M
cxfir32x32_alloc	80+16*M	72+8*M
cxfir32x32ep_alloc	80+16*M	72+8*M
cxfirf_alloc	64+16*M	72+8*M

Object initialization

```
cxfir16x16_handle_t cxfir16x16_init(void * objmem,
                          int M, int extIR, const complex_fract16 *  h)
cxfir24x24_handle_t cxfir24x24_init(void * objmem,
                          int M, int extIR, const complex_fract32 *  h)
cxfir32x16_handle_t cxfir32x16_init(void * objmem,
                          int M, int extIR, const complex_fract16 *  h)
cxfir32x32_handle_t cxfir32x32_init(void * objmem,
                          int M, int extIR, const complex_fract32 *  h)
cxfir32x32ep_handle_t cxfir32x32ep_init(void * objmem,
                          int M, int extIR, const complex_fract32 *  h)
cxfirf_handle_t cxfirf_init(void * objmem,
                          int M, int extIR, const complex_float *  h)
```

Type	Name	Size	Description
Input			
void*	objmem		allocated memory block
complex_fract32, complex_fract16, complex_float	h	M	complex filter coefficients; h[0] is to be multiplied with the newest sample , Q31, Q15 or floating point
int	M		length of filter
int	extIR		if zero, IR is copied from original location, otherwise not but user should keep alignment, order of coefficients and zero padding requirements shown below

Returns: handle to the object

Alignment, ordering and zero padding for external IR (extIR!=0)

Function	Alignment, bytes	Left zero padding, bytes	Coefficient order	Right zero padding, bytes
cxfir16x16_alloc, HiFi3	8	4	inverted	4
cxfir16x16_alloc, HiFi3z/4	8	2 before each copy	inverted: conjugated copy and (imaginary; real) copy at 4*(M+4) bytes offset	6 after each copy
cxfir32x16_init	8	4	inverted	4
cxfir24x24_init	8	0	inverted	0

cxfir32x32_init	8	0	inverted and conjugated	0
cxfir32x32ep_init	8	0	inverted and conjugated	0
cxfirf_init	8	0	direct	0

Update the delay line and compute filter output

```
void cxfir16x16_process(
            cxfir16x16_handle_t handle,
            complex_fract16 *  y,
            const complex_fract16*  x, int N );
void cxfir24x24_process(
            cxfir24x24_handle_t handle,
            complex_fract32 *  y,
            const complex_fract32*  x, int N );
void cxfir32x16_process(cxfir32x16_handle_t handle,
            complex_fract32 *  y,
            const complex_fract32 *  x, int N );
void cxfir32x32_process(cxfir32x32_handle_t handle,
            complex_fract32 *  y,
            const complex_fract32 *  x, int N );
void cxfir32x32ep_process(cxfir32x32ep_handle_t handle,
            complex_fract32 *  y,
            const complex_fract32 *  x, int N );
void cfirf_process (  cxfirf_handle_t handle,
              complex_float * y, const complex_float * x, int N);
```

Type	Name	Size	Description
Input			
complex_fract16, complex_fract32, complex_float	x	N	input samples , Q15, Q31 or floating point
int	N		length of sample block
Output			
complex_fract16, complex_fract32, complex_float	y	N	output samples , Q15, Q31 or floating point

Returns: none

Restrictions

`x,y` – should not overlap
`x,h` - aligned on a 8-bytes boundary
`N,M` - multiples of 4

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 `y`. The filter calculates `N` output samples from `N*D` input samples using `M` coefficients, requires last `M-1` samples on the delay line and updated in circular manner for each new `D` samples.
NOTE:
To avoid aliasing IR should be synthesized in such a way to be narrower than input sample rate divided to 2D.

Precision

6 variants available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit outputs
24x24	24-bit data, 24-bit coefficients, 24-bit outputs. Available for HiFi3/HiFi3z cores only
32x16	32-bit data, 16-bit coefficients, 32-bit outputs
32x32	32-bit data, 32-bit coefficients, 32-bit outputs
32x32ep	32-bit data, 32-bit coefficients, 32-bit outputs, use 72-bit accumulators for intermediate computations. Available for HiFi4 only
f	floating point. Requires VFPU core option

Algorithm

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

NOTE:
This is formal description of algorithm, in reality processing in done using circular buffers, so user application is not responsible for management of delay lines

Object allocation

```
size_t firdec16x16_alloc(int D, int M)
size_t firdec24x24_alloc(int D, int M)
size_t firdec32x16_alloc(int D, int M)
size_t firdec32x32_alloc(int D, int M)
size_t firdec32x32ep_alloc(int D, int M)size_t firdecf_alloc    (int D, int M)
```

Type	Name	Size	Description
Input			
int	D		decimation factor
int	M		length of filter

Returns: size of memory in bytes to be allocated

NOTE:
Approximate amount of requested memory is listed below

Function	Approximate memory requirements, bytes
firdec32x16_alloc	40+(M+8*D)*4+(M+4)*2
firdec16x16_alloc	40+(M+8*D)*2+(M+4)*2
firdec24x24_alloc	40+(M+8*D)*4+(M+4)*4
firdec32x32_alloc	40+(M+8*D)*4+(M+4)*4
firdec32x32ep_alloc	40+(M+8*D)*4+(M+4)*4
firdecf_alloc	40+(M+8*D)*4+(M+4)*4

Object initialization

```
firdec16x16_handle_t firdec16x16_init(void * objmem,
                          int D, int M, const int16_t *  h)
firdec24x24_handle_t firdec24x24_init(void * objmem,
                          int D, int M, const f24 *  h)
firdec32x16_handle_t firdec32x16_init(void * objmem,
                          int D, int M, const int16_t *  h)
firdec32x32_handle_t firdec32x32_init(void * objmem,
                          int D, int M, const int32_t *  h)
firdec32x32ep_handle_t firdec32x32ep_init(void * objmem,
                          int D, int M, const int32_t *  h)
firdecf_handle_t firdecf_init(void * objmem,
                          int D, int M, const float32_t *  h)
```

Type	Name	Size	Description
Input			
void*	objmem		allocated memory block
f24, int32_t, int16_t, float32_t	h	M	filter coefficients; h[0] is to be multiplied with the newest sample, Q31, Q15 or floating point
int	D		decimation factor
int	M		length of filter

Returns: handle to the object

Update the delay line and compute decimator output

```
void firdec16x16_process(firdec16x16_handle_t handle,
                         int16_t *  y, const int16_t * x, int N );
void firdec24x24_process(firdec24x24_handle_t handle,
                         f24 *  y, const f24 * x, int N );
void firdec32x16_process(firdec32x16_handle_t handle,
                         int32_t *  y, const int32_t * x, int N );
void firdec32x32_process(firdec32x32_handle_t handle,
                         int32_t *  y, const int32_t * x, int N );
void firdec32x32ep_process(firdec32x32ep_handle_t handle,
                         int32_t *  y, const int32_t * x, int N );
void firdecf_process (firdecf_handle_t handle,
                      float32_t * y, const float32_t * x, int N);
```

Type	Name	Size	Description
Input			
`int16_t, f24, int32_t, float32_t`	`x`	`D*N`	input samples , Q15, Q31 or floating point
`int`	`N`		length of output sample block, should be a multiple of 8
Output			
`int16_t, f24, int32_t, float32_t`	`y`	`N`	output samples, Q15, Q31 or floating point

Returns: none

Restrictions

`x,h,r` should not overlap
`x, h` - aligned on a 8-bytes boundary
`N` – multiple of 8
`D >1`

Conditions for optimum performance

`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 `y`. The filter calculates `N*D` output samples using `M*D` coefficients from `N` inputs. Delay line holds `M*D-1` last samples and updated in circular manner for each new sample.

Precision

6 variants available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit outputs
24x24	24-bit data, 24-bit coefficients, 24-bit outputs. Available for HiFi3/HiFi3z cores only
32x16	32-bit data, 16-bit coefficients, 32-bit outputs
32x32	32-bit data, 32-bit coefficients, 32-bit outputs
32x32ep	32-bit data, 32-bit coefficients, 32-bit outputs, use 72-bit accumulators for intermediate computations. Available for HiFi4 only
f	floating point. Requires VFPU core option

Algorithm

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

NOTE:
This is formal description of algorithm, in reality processing in done using circular buffers, so user application is not responsible for management of delay lines

Object allocation

```
size_t firinterp16x16_alloc(int D, int M)
size_t firinterp24x24_alloc(int D, int M)
size_t firinterp32x16_alloc(int D, int M)
size_t firinterp32x32_alloc(int D, int M)
size_t firinterp32x32ep_alloc(int D, int M)
size_t firinterpf_alloc    (int D, int M)
```

Type	Name	Size	Description
Input			
`int`	`D`		interpolation ratio
`int`	`M`		length of subfilter. Total length of filter is `M*D`

Returns: size of memory in bytes to be allocated
NOTE:
Approximate amount of requested memory is listed below

Function	Approximate memory requirements, bytes
`firinterp16x16_alloc`	`40+(M+8)*2+(M+4)*D*2`
`firinterp32x16_alloc`	`40+(M+8)*4+(M+4)*D*2`
`firinterp24x24_alloc`	`40+(M+8)*4+(M+4)*D*4`
`firinterp32x32_alloc`	`40+(M+8)*4+(M+4)*D*4`
`firinterp32x32ep_alloc`	`40+(M+8)*4+(M+4)*D*4`
`firinterpf_alloc`	`40+(M+8)*4+(M+4)*D*4`

Object initialization

```
firinterp16x16_handle_t firinterp16x16_init(void * objmem,
                          int D, int M, const int16_t*  h)
firinterp24x24_handle_t firinterp24x24_init(void * objmem,
                          int D, int M, const f24 *  h)
firinterp32x16_handle_t firinterp32x16_init(void * objmem,
                          int D, int M, const int16_t *  h)
firinterp32x32_handle_t firinterp32x32_init(void * objmem,
                          int D, int M, const int32_t *  h)
firinterp32x32ep_handle_t firinterp32x32ep_init(void * objmem,
                          int D, int M, const int32_t *  h)
firinterpf_handle_t firinterpf_init(void * objmem,
                          int D, int M, const float32_t *  h)
```

Type	Name	Size	Description
Input			
`void*`	`objmem`		allocated memory block
`f24, int32_t, int16_t, float32_t`	`h`	`M*D`	filter coefficients; h[0] is to be multiplied with the newest sample,Q31, Q15 or floating point
`int`	`D`		interpolation ratio
`int`	`M`		length of subfilter. Total length of filter is `M*D`

Returns: handle to the object

Update the delay line and compute interpolator output

```
void firinterp16x16_process(firinterp16x16_handle_t handle,
                            int16_t *  y, const int16_t *  x, int N);
void firinterp24x24_process(firinterp24x24_handle_t handle,
                            f24 *  y, const f24 *  x, int N);
void firinterp32x16_process(firinterp32x16_handle_t handle,
                            int32_t*  y, const int32_t*  x, int N);
void firinterp32x32_process(firinterp32x32_handle_t handle,
                            int32_t*  y, const int32_t*  x, int N);
void firinterp32x32ep_process(firinterp32x32ep_handle_t handle,
                            int32_t*  y, const int32_t*  x, int N);
void firinterpf_process     (firinterpf_handle_t handle,
                            float32_t * y, const float32_t * x, int N);
```

Type	Name	Size	Description
Input			
int16_t, f24, int32_t, float32_t	x	N	input samples, Q15, Q31 or floating point
int	N		length of input sample block
Output			
int16_t, f24, int32_t, float32_t	y	N*D	output samples, Q15, Q31 or floating point

Returns: none

Restrictions

x,h,y should not overlap
x,h - aligned on a 8-bytes boundary
M - multiples of 4
N - multiples of 8
D should be >1

Conditions for optimum performance

D - 2, 3 or 4

2.1.6 Circular convolution

Description

Performs circular convolution between vectors x (of length N) and y (of length M) resulting in vector r of length N.
Two versions of these functions available: faster version (fir_convol16x16, fir_convol24x24, fir_convol32x16, fir_convol32x32, fir_convol32x32ep, cxfir_convol32x16, fir_convolf) with some restrictions on input arguments and slower version (fir_convola16x16,fir_convola24x24, fir_convola32x16, fir_convola32x32, fir_convola32x32ep, cxfir_convola32x16, fir_convolaf) for arbitrary arguments. In addition, these slower version implementations require scratch memory area.

Precision

6 variants available:

Type	Description
16x16	16x16-bit data, 16-bit outputs
24x24	24x24-bit data, 24-bit outputs. Available for HiFi3/HiFi3z cores only
32x16	32x16-bit data, 32-bit outputs (both real and complex)
32x32	32x32-bit data, 32-bit outputs
32x32ep	32-bit data, 32-bit outputs, use 72-bit accumulators for intermediate computations. Available for HiFi4 only
f	floating point. Requires VFPU core option

Algorithm

$$r_k = \sum_{m=0}^{M-1} x_{\mathrm{mod}(k-m,N)} y_m, k = \overline{0...(N-1)}$$

Prototype

```
void fir_convol16x16  ( int16_t *  r, const int16_t *  x,const int16_t *  y,
                        int N, int M);
void fir_convol24x24  ( f24 *  r, const f24 *  x, const f24 *  y,
                        int N,  int M);
void fir_convol32x16  ( int32_t *  r, const int32_t *  x, const int16_t *  y,
                        int N, int M);
void fir_convol32x32  ( int32_t *  r, const int32_t *  x, const int32_t *  y,
                        int N, int M);
void fir_convol32x32ep( int32_t *  r, const int32_t *  x, const int32_t *  y,
                        int N, int M);
void cxfir_convol32x16( complex_fract32 *  r,
                        const complex_fract32 *  x, const complex_fract16 *  y,
                        int N,int M);
void fir_convolf      ( float32_t *  r,
                        const float32_t *  x, const float32_t *  y,
                        int N, int M);

void fir_convola16x16 ( void *  s,
                        int16_t  *  r, const int16_t  *  x, const int16_t  *  y,
                        int N, int M);
void fir_convola24x24 ( void *  s,
                        f24  *  r, const f24  *  x, const f24  *  y,
                        int N, int M);
void fir_convola32x16 ( void    *  s,
                        int32_t *  r, const int32_t *  x, const int16_t *  y,
                        int N, int M);
void fir_convola32x32 ( void    *  s,
                        int32_t *  r, const int32_t *  x, const int32_t *  y,
                        int N, int M);
void fir_convola32x32ep(void    *  s,
                        int32_t *  r, const int32_t *  x, const int32_t *  y,
                        int N, int M);
void cxfir_convola32x16(void    *  s,
                        complex_fract32 *  r,
                        const complex_fract32 *  x, const complex_fract16 *  y,
                        int N, int M);
void fir_convolaf      (void       *  s,
                        float32_t  *  r,
                        const float32_t  *  x, const float32_t  *  y,
                        int N, int M);
```

Arguments

Type	Name	Size	Description
Input			
int16_t, f24, int32_t, complex_fract32, float32_t	x	N	input data (Q15, Q31 or floating point)
f24, int16_t, complex_fract16, or float32_t	y	M	input data (Q31, Q15 or floating point)
int	N		length of x
int	M		length of y
Output			
int16_t, f24, int32_t, complex_fract32, float32_t	r	N	output data, Q15, Q31 or floating point
Temporary			
void	s		Scratch memory, FIR_CONVOLA16X16_SCRATCH_SIZE(N, M) FIR_CONVOLA24X24_SCRATCH_SIZE(N, M) FIR_CONVOLA32X16_SCRATCH_SIZE(N, M) FIR_CONVOLA32X32_SCRATCH_SIZE(N, M) FIR_CONVOLA32X32EP_SCRATCH_SIZE(N, M) CXFIR_CONVOLA32X16_SCRATCH_SIZE(N,M) FIR_CONVOLAF_SCRATCH_SIZE(N, M) bytes

Returned value none

Restrictions For slow versions (`fir_convola16x16, fir_convola24x24, fir_convola32x16, fir_convola32x32, fir_convola32x32ep, cxfir_convola32x16, fir_convolaf`):
`x,y,r,s` should not overlap
`s` should be aligned on 8-byte boundary
`N>=M-1`

For fast versions (`fir_convol16x16, fir_convol24x24, fir_convol32x16, fir_convol32x32, fir_convol32x32ep, cxfir_convol32x16, fir_convolf`):
`x,y,r` should not overlap
`x,y,r` should be aligned on 8-byte boundary
`N,M` – multiples of 4

2.1.7 Linear Convolution

Description Functions perform linear convolution between vectors `x` (of length `N`) and `y` (of length `M`) resulting in vector `r` of length `N+M-1`.

Precision 2 variants available:

Type	Description
16x16	16x16-bit data, 16-bit outputs
32x32	32x32-bit data, 32-bit outputs

Algorithm

$$r_k = \sum_{j=\max(k-M+1,0)}^{\min(N-1,k)} x_j y_{k-j}, k = \overline{0...(M+N-2)}$$

Prototype

```
void fir_lconvola16x16 (void    *  s,
                        int16_t *  r,
                  const int16_t *  x, const int16_t  *  y, int N, int M);
void fir_lconvola32x32 (void    *  s,
                        int32_t *  r,
                  const int32_t *  x, const int32_t  *  y, int N, int M);
```

Arguments

Type	Name	Size	Description
Input			
`int16_t, int32_t`	`x`	`N`	input data (Q15, Q31)
`int16_t, int32_t`	`y`	`M`	input data (Q31, Q15)
`int`	`N`		length of `x`
`int`	`M`		length of `y`
Output			
`int16_t, int32_t`	`r`	`M+N-1`	output data, Q15, Q31
Temporary			
`void`	`s`		Scratch memory, `FIR_LCONVOLA16X16_SCRATCH_SIZE( N, M)` `FIR_LCONVOLA32X32_SCRATCH_SIZE( N, M)` bytes

Returned value none

Restrictions `x,y,r,s` should not overlap
`s` should be aligned on 8-byte boundary
`N>0, M>0`
`N>=M-1`

2.1.8 Circular Correlation

Description

Estimates the circular cross-correlation between vectors x (of length N) and y (of length M) resulting in vector r of length N. It is a similar to correlation but y is read in opposite direction.
Two versions of these functions available: faster version (fir_xcorr16x16, fir_xcorr24x24, fir_xcorr32x16, fir_xcorr32x32, fir_xcorr32x32ep, fir_xcorrf, cxfir_xcorrf) with some restrictions on input arguments and slower version (fir_xcorra16x16, fir_xcorra24x24, fir_xcorra32x16, fir_xcorra32x32, fir_xcorra32x32ep, fir_xcorraf, cxfir_xcorraf) for arbitrary arguments. In addition, these slower version implementations require scratch memory area.

Precision

6 variants available:

Type	Description
16x16	16x16-bit data, 16-bit outputs
24x24	24x24-bit data, 24-bit outputs. Available for HiFi3/HiFi3z cores only
32x16	32x16-bit data, 32-bit outputs
32x32	32x32-bit data, 32-bit outputs
32x32ep	32-bit data, 32-bit outputs, use 72-bit accumulators for intermediate computations. Available for HiFi4 only
f	floating point (both real and complex data). Requires VFPU core option

Algorithm

$$r_k = \sum_{m=0}^{M-1} x_{\mathrm{mod}(k+m,N)} y_m, k = \overline{0...(N-1)}$$

Prototype

```
void fir_xcorr16x16 ( int16_t *  r, const int16_t *  x, const int16_t *  y,
                      int N, int M);
void fir_xcorr24x24 ( f24 *  r, const f24 *  x, const f24 *  y,
                      int N, int M);
void fir_xcorr32x16 ( int32_t *  r, const int32_t *  x, const int16_t *  y,
                      int N, nt M);
void fir_xcorr32x32 ( int32_t *  r, const int32_t *  x, const int32_t *  y,
                      int N, int M);
void fir_xcorr32x32ep(int32_t *  r, const int32_t *  x, const int32_t *  y,
                      int N, int M);
void fir_xcorrf      (float32_t *  r,
                      const float32_t *  x, const float32_t *  y,
                      int N, int M);
void cxfir_xcorrf    (complex_float *  r,
                      const complex_float *  x, const complex_float *  y,
                      int N, int M);

void fir_xcorra24x24 (void *  s,
                      int16_t *  r, const int16_t *  x, const int16_t *  y,
                      int N, int M);
void fir_xcorra24x24 (void *  s,
                      f24 *  r, const f24 *  x, const f24 *  y,
                      int N, int M);
void fir_xcorra32x16 (void    *  s,
                      int32_t *  r, const int32_t *  x, const int16_t *  y,
                      int N, int M);
void fir_xcorra32x32 (void    *  s,
                      int32_t *  r, const int32_t *  x, const int32_t *  y,
                      int N, int M);
void fir_xcorra32x32ep(void   *  s,
                      int32_t *  r, const int32_t *  x, const int32_t *  y,
                      int N, int M);
void fir_xcorraf     (void      *  s,
                      float32_t *  r, const float32_t * x, const float32_t * y,
                      int N, int M);
void cxfir_xcorraf   (void      *  s,
                      complex_float *  r,
                      const complex_float *  x, const complex_float *  y,
                      int N, int M);
```

Arguments

	Name	Size	Description
Input			
`int16_t, f24, int32_t, float32_t, complex_float`	`x`	`N`	input data (Q15, Q31 or floating point)
`f24, int16_t, float32_t, complex_float`	`y`	`M`	input data (Q31, Q15 or floating point)
`int`	`N`		length of `x`
`int`	`M`		length of `y`
Output			
`int16_t, f24, int32_t, float32_t, complex_float`	`r`	`N`	output data, Q15, Q31 or floating point
Temporary			
`void`	`s`		Scratch memory, `FIR_XCORRA16X16_SCRATCH_SIZE(N, M)` `FIR_XCORRA24X24_SCRATCH_SIZE(N, M)` `FIR_XCORRA32X16_SCRATCH_SIZE(N,M)` `FIR_XCORRA32X32_SCRATCH_SIZE(N, M)` `FIR_XCORRA32X32EP_SCRATCH_SIZE(N, M)` `FIR_XCORRAF_SCRATCH_SIZE(N, M)` `CXFIR_XCORRAF_SCRATCH_SIZE(N, M)` bytes

Returned value

none

Restrictions

For slow versions (`fir_xcorra16x16, fir_xcorra24x24, fir_xcorra32x16, fir_xcorra32x32ep, fir_xcorra32x32, fir_xcorraf, cxfir_xcorraf`):
`x,y,r,s` should not overlap
`s` should be aligned on 8-byte boundary
`N>=M-1`

For fast versions (`fir_xcorr16x16, fir_xcorr24x24, fir_xcorr32x16, fir_xcorr32x32, fir_xcorr32x32ep, fir_xcorrf, cxfir_xcorrf`):
`x,y,r` should not overlap
`x,y,r` should be aligned on 8-byte boundary
`N,M` – multiples of 4

2.1.9 Linear Correlation

Description

Functions estimate the linear cross-correlation between vectors `x` (of length `N`) and `y` (of length `M`) resulting in vector `r` of length `N+M-1`. It is a similar to convolution but `y` is read in opposite direction.

Precision

2 variants available:

Type	Description
16x16	16x16-bit data, 16-bit outputs
32x32	32x32-bit data, 32-bit outputs

Algorithm

$$r_k = \sum_{j=\max(k-M+1,0)}^{\min(N-1,k)} x_j y^*_{M-1-(k-j)}, k = \overline{0...(M+N-2)}$$

Prototype

```
void fir_lxcorra16x16 ( void    *  s,
                        int16_t *  r,
                  const int16_t *  x, const int16_t  *  y, int N, int M);
void fir_lxcorra32x32 ( void    *  s,
                        int32_t *  r,
                  const int32_t *  x, const int32_t  *  y, int N, int M);
```

Arguments

Type	Name	Size	Description
Input			
`int16_t, int32_t`	`x`	`N`	input data (Q15, Q31)
`int16_t, int32_t`	`y`	`M`	input data (Q31, Q15)
`int`	`N`		length of `x`
`int`	`M`		length of `y`
Output			
`int16_t, int32_t,`	`r`	`M+N-1`	output data, Q15, Q31
Temporary			
`void`	`s`		Scratch memory, `FIR_LXCORRA16X16_SCRATCH_SIZE(N, M)` `FIR_LXCORRA32X32_SCRATCH_SIZE(N, M)` bytes

Returned value

none

Restrictions

`x,y,r,s` should not overlap
`s` should be aligned on 8-byte boundary
`N>0, M>0`
`N>=M-1`

2.1.10 Circular Autocorrelation

Description

Estimates the auto-correlation of vector `x`. Returns autocorrelation of length `N`.
Two versions of these functions available: faster version (`fir_acorr16x16, fir_acorr24x24, fir_acorr32x32, fir_acorr32x32ep, fir_acorrf`) with some restrictions on input arguments and slower version (`fir_acorra16x16, fir_acorra24x24, fir_acorra32x32, fir_acorra32x32ep, fir_acorraf`) for arbitrary arguments. In addition, this slower version implementations require scratch memory area.

Precision

5 variants available:

Type	Description
16x16	16-bit data, 16-bit outputs
24x24	24-bit data, 24-bit outputs. Available for HiFi3/HiFi3z cores only
32x32	32-bit data, 32-bit outputs
32x32ep	32-bit data, 32-bit outputs, use 72-bit accumulators for intermediate computations. Available for HiFi4 only
f	floating point. Requires VFPU core option

Algorithm

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

Prototype

```
void fir_acorr16x16  ( int16_t  *  r, const int16_t  *  x, int N);
void fir_acorr24x24  ( f24      *  r, const f24      *  x, int N);
void fir_acorr32x32  ( int32_t  *  r, const int32_t  *  x, int N);
void fir_acorr32x32ep( int32_t  *  r, const int32_t  *  x, int N);
void fir_acorrf      ( float32_t*  r, const float32_t*  x, int N);

void fir_acorra16x16  (void* s, int16_t  *  r, const int16_t  *  x, int N);
void fir_acorra24x24  (void* s, f24      *  r, const f24      *  x, int N);
void fir_acorra32x32  (void* s, int32_t  *  r, const int32_t  *  x, int N);
void fir_acorra32x32ep(void* s, int32_t  *  r, const int32_t  *  x, int N);
void fir_acorraf      (void* s, float32_t*  r, const float32_t*  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int16_t, int32_t, f24 or float32_t`	`x`	`N`	input data (Q15, Q31 or floating point)
`int`	`N`		length of `x`
Output			
`int16_t, int32_t, f24 or float32_t`	`r`	`N`	output data, Q15, Q31 or floating point
Temporary			
`void`	`s`		Scratch memory, `FIR_ACORRA16X16_SCRATCH_SIZE( N )` `FIR_ACORRA24X24_SCRATCH_SIZE( N )` `FIR_ACORRA32X32_SCRATCH_SIZE( N )FIR_ACORRAF_SCRATCH_SIZE( N )` bytes

Returned value

none

Restrictions

For slow versions (`fir_acorr16x16, fir_acorr24x24, fir_acorr32x32, fir_acorr32x32ep, fir_acorrf`):
`x,r,s` should not overlap
`N` - must be non-zero
`s` - aligned on an 8-bytes boundary
For fast versions (`fir_acorra16x16, fir_acorra24x24, fir_acorra32x32, fir_acorra32x32ep, fir_acorraf`):
`x,r` should not overlap
`x,r` should be aligned on 8-byte boundary
`N` – non-zero multiple of 4

2.1.11 Linear Autocorrelation

Description

Functions estimate the linear auto-correlation of vector `x`. Returns autocorrelation of length `N`.

Precision

2 versions available:

Type	Description
16x16	16-bit data, 16-bit outputs
32x32	32-bit data, 32-bit outputs

Algorithm

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

Prototype

```
void fir_lacorra16x16 (void* s, int16_t  *  r, const int16_t  *  x, int N);
void fir_lacorra32x32 (void* s, int32_t  *  r, const int32_t  *  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int16_t, int32_t`	`x`	`N`	input data (Q15, Q31)
`int`	`N`		length of `x`
Output			
`int16_t, int32_t`	`r`	`N`	output data, Q15, Q31
Temporary			
`void`	`s`		Scratch memory, `FIR_LACORRA16X16_SCRATCH_SIZE( N )` `FIR_LACORRA32X32_SCRATCH_SIZE( N )` bytes

Returned value none

Restrictions `x,r,s` should not overlap
`N>0`
`s` - aligned on an 8-bytes boundary

2.1.12 Blockwise Adaptive LMS Algorithm for Real Data

Description

Blockwise LMS algorithm performs filtering of reference samples `x[N+M-1]`, computation of error `e[N]` over a block of input samples `r[N]` and makes blockwise update of IR to minimize the error output. Algorithm includes FIR filtering, calculation of correlation between the error output `e[N]` and reference signal `x[N+M-1]` and IR taps update based on that correlation.
NOTES:
1. The algorithm must be provided with the normalization factor, which is the power of the reference signal times `N` - the number of samples in a data block. This can be calculated using the `vec_power24x24()` or `vec_power16x16()` function. In order to avoid the saturation of the normalization factor, it may be biased, i.e. shifted to the right. If it's the case, then the adaptation coefficient must be also shifted to the right by the same number of bit positions.
2. This algorithm consumes less CPU cycles per block than single sample algorithm at similar convergence rate.
3. Right selection of `N` depends on the change rate of impulse response: on static or slow varying channels convergence rate depends on selected `mu` and `M`, but not on `N`.
4. 16x16 routine may converge slower on small errors due to roundoff errors. In that cases, 16x32 routine will give better results although convergence rate on bigger errors is the same

Precision

6 variants available:

Type	Description
16x16	16-bit coefficients, 16-bit data, 16-bit output
24x24	24-bit coefficients, 24-bit data, 24-bit output. Available for HiFi3/HiFi3z cores only
16x32	32-bit coefficients, 16-bit data, 16-bit output
32x32	32-bit coefficients, 32-bit data, 32-bit output
32x32ep	32-bit data, 32-bit coefficients, 32-bit outputs, use 72-bit accumulators for intermediate computations. Available for HiFi4 only
f	floating point. Requires VFPU core option

Algorithm

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

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

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

Prototype

```
void fir_blms16x16 (  int16_t*  e, int16_t *  h,
                      const int16_t *  r,
                      const int16_t *  x,
                      int16_t norm, int16_t   mu,
                      int   N,   int   M);
void fir_blms24x24 (  f24 *  e, f24 *  h,
                      const f24 *  r,
                      const f24 *  x,
                      f24   norm,f24   mu,
                      int   N,   int   M);
void fir_blms16x32 (  int32_t *  e, int32_t *  h,
                      const int16_t *  r,
                      const int16_t *  x,
                      int32_t   norm,int16_t   mu,
                      int       N,   int       M);
void fir_blms32x32 (  int32_t *  e, int32_t *  h,
                      const int32_t *  r,
                      const int32_t *  x,
                      int32_t   norm, int32_t mu,
                      int       N,   int       M);
void fir_blmsf      ( float32_t * e, float32_t * h, const float32_t * r,
                      const float32_t * x,
                      float32_t norm, float32_t mu,
                      int          N, int       M );
```

Arguments

Type	Name	Size	Description
Input			
int16_t, f24, int32_t, float32_t	h	M	impulse response, Q15, Q31 or floating point
f24, int16_t, int32_t or float32_t	r	N	reference (near end) data vector. First in time value is in r[0], Q31, Q15 or floating point
f24, int16_t, int32_t or float32_t	x	N+M-1	input (far end) data vector. First in time value is in x[0], Q31, Q15 or floating point
int16_t, f24, int32_t, float32_t	norm		normalization factor: power of signal multiplied by N, Q31, Q15 or floating point
f24, int16_t int32_t, float32_t	mu		adaptation coefficient in Q31, Q15 or floating point (LMS step)
int	N		length of data block
int	M		length of h
Output			
f24, int16_t, int32_t, float32_t	e	N	estimated error, Q31, Q15 or floating point
f24, int16_t, int32_t, float32_t	h	M	updated impulse response, Q31, Q15 or floating point

Returned value none

Restrictions

`h,x,r,y,e` – should not overlap
`x,e,h,r` - aligned on a 8-bytes boundary
`N,M` - multiples of 8

2.2 IIR filters

2.2.1 Bi-quad Real Block IIR

Description

Computes a real IIR filter (cascaded IIR direct form I or II using 5 coefficients per bi-quad + gain term) . Input data are stored in vector `x`. Filter output samples are 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, typically biquad stages have big peaks in their step response which may cause undesirable overflows at the intermediate outputs. To avoid that the additional scale factors `coef_g[M]` may be applied. These per-section scale factors may require some tuning to find a compromise between quantization noise and possible overflows. Output of the last section is directed to an additional multiplier, with the gain factor being a power of two, either negative or non-negative. It is specified through the total gain shift amount parameter `gain` of each filter initialization function
- 16x16 filters may suffer more from accumulation of the roundoff errors, so filters should be properly designed to match noise requirements

Precision

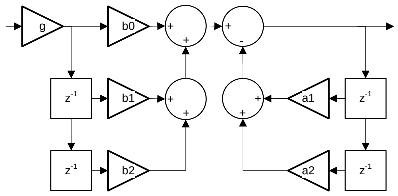
5 variants available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit intermediate stage outputs (DF1, DF II form)
24x24	32-bit data, 24-bit coefficients, 32-bit intermediate stage outputs. Available for HiFi3/HiFi3z cores only
32x16	32-bit data, 16-bit coefficients, 32-bit intermediate stage outputs
32x32	32-bit data, 32-bit coefficients, 32-bit intermediate stage outputs (DF I, DF II form)
f	floating point (DF I, DF II and DF IIt). Requires VFPU core option

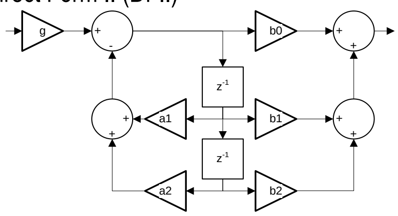
Algorithm

A block of N real input samples is sequentially passed through M bi-quad sections. There are two options for the implementation structure of a single section:

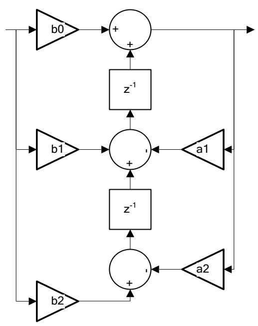
Direct Form I (DFI)



Direct Form II (DFII)



Direct Form II transposed (DF IIt)



Object allocation

```
size_t bqriir16x16_df1_alloc(int M)
size_t bqriir16x16_df2_alloc(int M)
size_t bqriir24x24_df1_alloc(int M)
size_t bqriir24x24_df2_alloc(int M)
size_t bqriir32x16_df1_alloc(int M)
size_t bqriir32x16_df2_alloc(int M)
size_t bqriir32x32_df1_alloc(int M)
size_t bqriir32x32_df2_alloc(int M)
size_t bqriirf_df1_alloc(int M)
size_t bqriirf_df2_alloc(int M)
size_t bqriirf_df2t_alloc(int M)
size_t bqciirf_df1_alloc(int M)
```

Type	Name	Size	Description
Input			
`int`	`M`		number of bi-quad sections

Returns: size of memory in bytes to be allocated

Object initialization

```
bqriir16x16_df1_handle_t bqriir16x16_df1_init(void * objmem, int M,
           const int16_t * coef_sos, const int16_t * coef_g, int16_t gain );
bqriir16x16_df2_handle_t bqriir16x16_df2_init(void * objmem, int M,
           const int16_t * coef_sos, const int16_t * coef_g,  int16_t gain);
bqriir24x24_df1_handle_t bqriir24x24_df1_init(void * objmem, int M,
           const f24     * coef_sos, const int16_t * coef_g, int16_t gain );
bqriir24x24_df2_handle_t bqriir24x24_df2_init(void * objmem, int M,
           const f24     * coef_sos,const int16_t * coef_g,int16_t gain);
bqriir32x16_df1_handle_t bqriir32x16_df1_init(void * objmem, int M,
           const int16_t * coef_sos, const int16_t * coef_g, int16_t gain);
bqriir32x16_df2_handle_t bqriir32x16_df2_init(void * objmem, int M,
           const int16_t * coef_sos, const int16_t * coef_g, int16_t gain);
bqriir32x32_df1_handle_t bqriir32x32_df1_init(void * objmem, int M,
           const int32_t * coef_sos, const int16_t * coef_g, int16_t gain)
bqriir32x32_df2_handle_t bqriir32x32_df2_init(void * objmem, int M,
           const int32_t * coef_sos, const int16_t * coef_g, int16_t gain)
bqriirf_df1_handle_t bqriirf_df1_init(void * objmem, int M,
           const float32_t* coef_sos, int16_t gain );
bqriirf_df2_handle_t bqriirf_df2_init(void * objmem, int M,
           const float32_t * coef_sos, int16_t gain);
bqriirf_df2t_handle_t bqriirf_df2t_init(void * objmem, int M,
           const float32_t * coef_sos, int16_t gain);
bqciirf_df1_handle_t bqciirf_df1_init(void * objmem, int M,
           const float32_t * coef_sos, int16_t gain);
```

Type	Name	Size	Description
Input			
`void*`	`objmem`		allocated memory block
`int`	`M`		number of bi-quad sections
`f24`, `int32_t`, `int16_t`, `float32_t`	`coef_sos`	`M*5`	filter coefficients stored in blocks of 5 numbers: `b0 b1 b2 a1 a2`. For fixed-point functions, fixed point format of filter coefficients is Q1.14 for 16x16 and 32x16, or Q1.30 for 32x32 and 24x24 (in the latter case 8 LSBs are actually ignored).
`int16_t`	`coef_g`	`M`	scale factor for each section, Q15 (for fixed-point functions only). Please note that 24x24 DFI implementation internally truncates scale factors to Q7 values.
`int16_t`	`gain`		total gain shift amount, -48..15

Returns: handle to the object

Update the delay line and compute filter output

```
void bqriir16x16_df1(bqriir16x16_df1_handle_t _bqriir,
                void *  s,int16_t *  r,const int16_t *x, int N);
void bqriir16x16_df2(bqriir16x16_df2_handle_t _bqriir,
                void *  s,int16_t *  r,const int16_t *x, int N);
void bqriir24x24_df1(bqriir24x24_df1_handle_t _bqriir,
                void *  s,int32_t *  r,const int32_t *x, int N);
void bqriir24x24_df2(bqriir24x24_df2_handle_t _bqriir,
                void *  s,int32_t *  r,const int32_t *x, int N);
void bqriir32x16_df1(bqriir32x16_df1_handle_t _bqriir,
                void *  s,int32_t *  r,const int32_t *x, int N);
void bqriir32x16_df2(bqriir32x16_df2_handle_t _bqriir,
                void *  s,int32_t *  r,const int32_t *x, int N);
void bqriir32x32_df1(bqriir32x32_df1_handle_t _bqriir,
                void *  s,int32_t *  r,const int32_t *x, int N);
void bqriir32x32_df2(bqriir32x32_df2_handle_t _bqriir,
                void *  s,int32_t *  r,const int32_t *x, int N);
void bqriirf_df1 (bqriirf_df1_handle_t,
                      float32_t    * r, const float32_t * x, int N);
void bqriirf_df2 (bqriirf_df2_handle_t,
                      float32_t    * r, const float32_t * x, int N);
void bqriirf_df2t (bqriirf_df2t_handle_t,
                      float32_t    * r, const float32_t * x, int N);
void bqciirf_df1 (bqciirf_df1_handle_t,
                      complex_float* r, const complex_float * x, int N);
```

Type	Name	Size	Description
Input			
`int16_t`, `int32_t`, `float32_t`, `complex_float`	`x`	`N`	input samples, Q31, Q15
`int`	`N`		length of input sample block
Output			
`int16_t`, `int32_t`, `float32_t`, `complex_float`	`r`	`N`	output data, Q31, Q15 or floating point
Temporary			
`void*`	`s`		scratch memory area (for fixed-point functions only). Minimum number of bytes depends on selected filter structure and precision (see spreadsheet below) If a particular macro returns zero, then the corresponding IIR doesn't require a scratch area and parameter `s` may hold zero

Returns: none

Function	Scratch memory, bytes
`bqriir16x16_df1`	`BQRIIR16X16_DF1_SCRATCH_SIZE(N,M)`
`bqriir16x16_df2`	`BQRIIR16X16_DF2_SCRATCH_SIZE(N,M)`
`bqriir24x24_df1`	`BQRIIR24X24_DF1_SCRATCH_SIZE(N,M)`
`bqriir24x24_df2`	`BQRIIR24X24_DF2_SCRATCH_SIZE(N,M)`
`bqriir32x16_df1`	`BQRIIR32X16_DF1_SCRATCH_SIZE(N,M)`
`bqriir32x16_df2`	`BQRIIR32X16_DF2_SCRATCH_SIZE(N,M)`
`bqriir32x32_df1`	`BQRIIR32X32_DF1_SCRATCH_SIZE(N,M)`
`bqriir32x32_df2`	`BQRIIR32X32_DF2_SCRATCH_SIZE(N,M)`

Returned value

none

Restrictions

`x`,`r`,`s`,`coef_g`,`coef_sos` must not overlap
`N` - must be a multiple of 2
`s` - whenever supplied must be aligned on an 8-bytes boundary

2.2.2 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`. Input scaling is done before the first cascade for normalization and overflow protection.

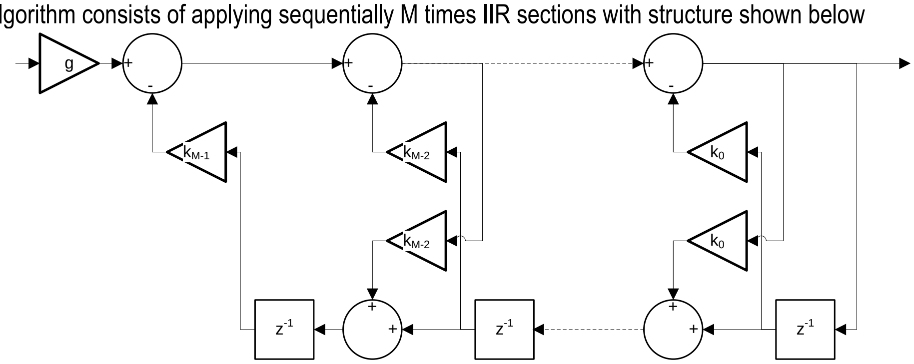
Precision

5 variants available:

Type	Description
16x16	16-bit data, 16-bit coefficients
24x24	24-bit data, 24-bit coefficients. Available for HiFi3/HiFi3z cores only
32x16	32-bit data, 16-bit coefficients
32x32	32-bit data, 32-bit coefficients
f	floating point. Requires VFPU core option

Algorithm

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



Object allocation

```
size_t latr16x16_alloc(int M);
size_t latr24x24_alloc(int M);
size_t latr32x16_alloc(int M);
size_t latr32x32_alloc(int M);
size_t latrf_alloc    (int M);
```

Type	Name	Size	Description
Input			
`int`	`M`		number of sections

Returns: size of memory in bytes to be allocated

Object initialization

```
latr16x16_handle_t latr16x16_init
   (void * objmem, int M,const int16_t    *  k, int16_t   scale);
latr24x24_handle_t latr24x24_init
   (void * objmem, int M,const f24        *  k, f24       scale);
latr32x16_handle_t latr32x16_init
   (void * objmem, int M, const int16_t   *  k, int16_t   scale);
latr32x32_handle_t latr32x32_init
   (void * objmem, int M, const int32_t   *  k, int32_t   scale);
latrf_handle_t     latrf_init
   (void * objmem, int M, const float32_t *  k, float32_t scale);
```

Type	Name	Size	Description
Input			
`void*`	`objmem`		allocated memory block
`int`	`M`		number of sections
`f24, int16_t, int32_t or float32_t`	`k`	`M`	reflection coefficients, Q31, Q15 or floating point
`f24, int16_t, int32_t or float32_t`	`scale`	`M`	input scale factor g, Q31, Q15 or floating point

Returns: handle to the object

Update the delay line and compute filter output

```
void latr16x16_process
    (latr16x16_handle_t handle, int16_t   * r, const int16_t   * x, int N);
void latr24x24_process
    (latr24x24_handle_t handle, f24       * r, const f24       * x, int N);
void latr32x16_process
    (latr32x16_handle_t handle, int32_t   * r, const int32_t   * x, int N);
void latr32x32_process
    (latr32x32_handle_t handle, int32_t   * r, const int32_t   * x, int N);
void latrf_process
    (latrf_handle_t     handle, float32_t * r, const float32_t * x, int N);
```

Type	Name	Size	Description
Input			
`int16_t, f24, int32_t or float32_t`	`x`	`N`	input samples, Q31, Q15 or floating point
`int`	`N`		length of input sample block
Output			
`int16_t, f24, int32_t or float32_t`	`r`	`N`	output data, Q31, Q15 or floating point

Returns: none

Returned value none

Restrictions `x,r,k` should not overlap

Conditions for optimum performance For optimum performance `M` should be in range `1…8`

2.3 Mathematics

A number of DSP Library functions supersede standard floating-point mathematical functions similar to defined in `<math.h>`, as listed below:

ANSI function	Scalar function	reference
atanf	scl_atanf	2.3.9
atan2f	scl_atan2f	2.3.10
cosf	scl_cosinef	2.3.7
sinf	scl_sinef	2.3.7
tanf	scl_tanf	2.3.8
logf	scl_lognf	2.3.3
log2f	scl_log2f	2.3.3
log10f	scl_log10f	2.3.3
expf	scl_antilognf	2.3.4
exp2f	scl_antilog2f	2.3.4
alog10f	scl_antilog10f	2.3.4

All these functions conform to ISO/IEC 9899 standard (commonly referred to as C99) in respect to function semantics, parameters and return value specification. Moreover, floating-point mathematical functions handle error conditions in a way that differs from general DSP Library approach as stated in 1.8. Aforementioned functions follow the next ground rules:

- Each function executes as if it were a single operation, and may generate any of “invalid”, “overflow” or “divide-by-zero” floating-point exceptions only to reflect the result of that operation.
- A domain error occurs if input argument(s) fall out of the function domain as defined in function specification. In such a case, the function assigns `EDOM` to the integer expression `errno`, raises the “invalid” floating-point exception, and returns a quiet `NaN`.
- `NaN` as an input argument is a special kind of domain error. Namely, the integer expression `errno` acquires `EDOM` and returned value is a quiet `NaN`, but the function raises the “invalid” floating-point exception only if the input argument is a *signaling* `NaN`.
- A floating-point result overflows if the magnitude of the mathematical result is finite but so large that the target floating-point type cannot represent the mathematical result without extraordinary round-off error (for example, `scl_antilognf(100.0f)`). If a function detects a floating-point result overflow, it assigns `ERANGE` to the integer expression `errno`, raises the “overflow” floating-point exception and returns the properly signed infinity value.

The set of floating-point mathematical functions conforming to ISO/IEC 9899 includes vectorized variants of all the functions listed above. Due to the performance reasons, these vectorized functions do not handle `errno` and may generate exceptions in bit different manner to minimize the overhead.

2.3.1 Reciprocal on Q31/Q15 Numbers

Description

These routines return the fractional and exponential portion of the reciprocal of a vector `x` of Q31 or Q15 numbers. Since the reciprocal is always greater than 1, it returns fractional portion `frac` in Q(31-`exp`) or Q(15-`exp`) format and exponent exp so true reciprocal value in the Q0.31/Q0.15 may be found by shifting fractional part left by exponent value.
NOTE: `scl_recip32x32()` uses packed output for mantissa/exponent. To take a full precision, just call vectorized counterpart.

Mantissa accuracy is 1 LSB, so relative accuracy is:

`vec_recip16x16, scl_recip16x16`	6.2e-5
`vec_recip24x24, scl_recip32x32, scl_recip24x24`	2.4e-7
`vec_recip32x32`	9.2e-10

Precision

3 variants available:

Type	Description
32x32	32-bit input, 32-bit output.
24x24	24-bit input, 24-bit output. Available for HiFi3/HiFi3z cores only
16x16	16-bit input, 16-bit output.

Algorithm

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

Prototype

```
void vec_recip32x32 (
                  int32_t *  frac,
                  int16_t *exp,
                  const int32_t *  x,
                  int N)
void vec_recip24x24 (
                  f24 *  frac,
                  int16_t *exp,
                  const f24 *  x,
                  int N)
void vec_recip16x16 (
                  int16_t *  frac,
                  int16_t *exp,
                  const int16_t *  x,
                  int N)
```

Arguments

Type	Name	Size	Description
Input			
`f24,int32_t or int16_t`	`x`	`N`	input data, Q31 or Q15
`int`	`N`		length of vectors
Output			
`f24, int32_t or int16_t`	`frac`	`N`	fractional part of result, Q(31-`exp`) or Q(15-`exp`)
`int16_t`	`exp`	`N`	exponent of result

Returned value

None

Restrictions

`x,frac,exp` should not overlap

Conditions for optimum performance

`frac,x` - aligned on 8-byte boundary
`N` - multiple of 4 and >4

Scalar versions

Prototype

```
uint32_t scl_recip32x32 (int32_t x)
uint32_t scl_recip24x24 (f24 x)
uint32_t scl_recip16x16 (int16_t x)
```

Arguments

Type	Name	Description
Input		
`f24, int32_t or int16_t`	`x`	input data, Q31 or Q15

Returned value

packed value:
`scl_recip24x24(),scl_recip32x32():`
bits 23…0 fractional part
bits 31…24 exponent
`scl_recip16x16():`
bits 15…0 fractional part
bits 31…16 exponent

2.3.2 Division of Q31/Q15 Numbers

Description

These routines perform pair-wise division of vectors written in Q31 or Q15 format. They return 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(31-`exp`) or Q(15-`exp`) format and exponent exp so true division result in the Q0.31 may be found by shifting fractional part left by exponent value.
For division to 0, the result is not defined

Two versions of routines are available: regular versions (`vec_divide32x32, vec_divide24x24, vec_divide16x16`) work with arbitrary arguments, faster versions (`vec_divide32x32_fast, vec_divide24x24_fast, vec_divide16x16_fast`) apply some restrictions.

NOTE: `scl_divide32x32()` uses packed output for mantissa/exponent. To take a full precision, just call vectorized counterpart.

Mantissa accuracy is 2 LSB, so relative accuracy is:

`vec_divide16x16, scl_divide16x16`	1.2e-4
`vec_divide24x24, scl_divide32x32, scl_divide24x24`	4.8e-7
`vec_divide32x32`	1.8e-9

Precision

4 variants available:

Type	Description
64x32i	integer division, 64-bit nominator, 32-bit denominator, 32-bit output
32x32	32-bit inputs, 32-bit output.
24x24	24-bit inputs, 24-bit output. Available for HiFi3/HiFi3z cores only
16x16	16-bit inputs, 16-bit output.

Algorithm

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

Prototype

```
void vec_divide64x32i
                (int32_t * frac, const int64_t * x, const int32_t * y, int N);
void vec_divide32x32
                (int32_t * frac, int16_t *exp,
                 const int32_t *  x, const int32_t *  y, int N)
void vec_divide24x24
                (f24 *  frac, int16_t *exp,
                 const f24 *  x, const f24 *  y, int N)
void vec_divide16x16
                (int16_t *  frac, int16_t *exp,
                 const int16_t *  x, const int16_t *  y, int N)
void vec_divide32x32_fast
                (int32_t *  frac, int16_t *exp,
                 const int32_t *  x, const int32_t *  y, int N);
void vec_divide24x24_fast
                (f24 *  frac, int16_t *exp,
                 const f24 *  x, const f24 *  y, int N) ;
void vec_divide16x16_fast
                (int16_t *  frac, int16_t *exp,
                 const int16_t *  x, const int16_t *  y, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int64_t, f24, int32_t or int16_t`	`x`	`N`	nominator, 64-bit integer, Q31 or Q15
`f24, int32_t or int16_t`	`y`	`N`	denominator, 32-bit integer, Q31 or Q15
`int`	`N`		length of vectors
Output			
`f24, int32_t or int16_t`	`frac`	`N`	fractional parts of result, Q(31-`exp`) or Q(15-`exp`)
`int16_t`	`exp`	`N`	exponents of result

Returned value

none

Restrictions

For regular versions (`vec_divide64x32i, vec_divide32x32, vec_divide24x24, vec_divide16x16`):
`x,y,frac,exp` should not overlap

For faster versions (`vec_divide32x32_fast, vec_divide24x24_fast, vec_divide16x16_fast`):
`x,y,frac,exp` should not overlap
`x, y, frac` to be aligned by 8-byte boundary
`N` - multiple of 4.

Scalar versions

Prototype

```
int32_t  scl_divide64x32(int64_t x,int32_t y);
uint32_t scl_divide32x32 (int32_t x, int32_t y)
uint32_t scl_divide24x24 (f24 x, f24 y)
uint32_t scl_divide16x16 (int16_t x, int16_t y)
```

Arguments

Type	Name	Description
Input		
`f24 or int16_t`	`x`	nominator, 64-bit integer, Q31 or Q15
`f24 or int16_t`	`y`	denominator, 32-bit integer, Q31 or Q15

Returned value

`scl_divide64x32()`	integer remainder
`scl_divide24x24(), scl_divide32x32()`	packed value: bits 23…0 fractional part, bits 31…24 exponent
`scl_divide16x16()`	packed value: bits 15…0 fractional part, bits 31…16 exponent

2.3.3 Logarithm

Description

Different kinds of logarithm (base 2, natural, base 10). 24 and 32-bit fixed point functions interpret input as Q16.15, represent results in Q6.25 format or return 0x80000000 on negative of zero input
Accuracy :

`vec_log2_32x32,scl_log2_32x32 , vec_log2_24x24,scl_log2_24x24`	`730 (2.2e-5)`
`vec_logn_32x32,scl_logn_32x32 , vec_logn_24x24,scl_logn_24x24`	`510 (1.5e-5)`
`vec_log10_32x32,scl_log10_32x32, vec_log10_24x24,scl_log10_24x24`	`230 (6.9e-6)`
floating point	`2 ULP`

NOTES:

1. although 32 and 24 bit functions provide the same accuracy, 32-bit functions have better input/output resolution (dynamic range)
2. Floating point functions are compatible with standard ANSI C routines and set `errno` and exception flags accordingly.
3. Floating point functions limit the range of allowable input values:

- If `x<0`, the result is set to `NaN`. In addition, scalar floating point functions assign the value `EDOM` to `errno` and raise the "invalid" floating-point exception.
- If `x==0`, the result is set to minus infinity. Scalar floating point functions assign the value `ERANGE` to `errno` and raise the "divide-by-zero" floating-point exception.

Precision

3 variants available:

Type	Description
32x32	32-bit inputs, 32-bit outputs
24x24	24-bit inputs, 24-bit outputs. Available for HiFi3/HiFi3z cores only
f	floating point. Requires VFPU core option

Algorithm

$$z_n = \log_K x_n, n = \overline{0 \ldots N-1}, K = 2, e, 10$$

Prototypes

```
void vec_log2_32x32 (  int32_t *  z, const int32_t *  x, int N);
void vec_logn_32x32 (  int32_t *  z, const int32_t *  x, int N);
void vec_log10_32x32(  int32_t *  z, const int32_t *  x, int N);
void vec_log2_24x24 (      f24 *  z, const f24 *  x, int N);
void vec_logn_24x24 (      f24 *  z, const f24 *  x, int N);
void vec_log10_24x24(      f24 *  z, const f24 *  x, int N);
void vec_log2f      (float32_t *  z, const float32_t *  x, int N);
void vec_lognf      (float32_t *  z, const float32_t *  x, int N);
void vec_log10f     (float32_t *  z, const float32_t *  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`f24, int32_t, float32_t`	`x`	`N`	input data, Q16.15 or floating point
`int`	`N`		length of vectors
Output			
`f24, int32_t, float32_t`	`z`	`N`	Q6.25 or floating point

Returned value

none

Restrictions

`x,z` – should not overlap

Scalar versions

Prototypes

```
int32_t scl_log2_32x32 (int32_t x);
int32_t scl_logn_32x32 (int32_t x);
int32_t scl_log10_32x32(int32_t x);
f24 scl_log2_24x24 (f24 x);
f24 scl_logn_24x24 (f24 x);
f24 scl_log10_24x24(f24 x);
float32_t scl_log2f (float32_t x);
float32_t scl_lognf  (float32_t x);
float32_t scl_log10f(float32_t x);
```

Arguments

Type	Name	Description
Input		
`f24, int32_t, float32_t`	`x`	input data, Q16.15 or floating point

Returned value

result, Q6.25 or floating point

2.3.4 Antilogarithm

Description

These routines calculate antilogarithm (base2, natural and base10). 24 and 32-bit fixed-point functions accept inputs in Q6.25 and form outputs in Q16.15 format and return 0x7FFFFFFF in case of overflow and 0 in case of underflow.

NOTES:
1. Although 32 and 24 bit functions provide the similar accuracy, 32-bit functions have better input/output resolution (dynamic range).
2. Floating point functions are compatible with standard ANSI C routines and set `errno` and exception flags accordingly.

Precision

3 variants available:

Type	Description
32x32	32-bit inputs, 32-bit outputs. Accuracy: 8e-6*y+1LSB
24x24	24-bit inputs, 24-bit outputs. Accuracy: 8e-6*y+1LSB. Available for HiFi3/HiFi3z cores only
f	floating point. Accuracy: 2 ULP. Requires VFPU core option

Algorithm

$$y_n = 2^{x_n}$$
$$y_n = e^{x_n}$$
$$y_n = 10^{x_n}$$

Prototype

```
void vec_antilog2_32x32(int32_t *  y, const int32_t*  x, int N);
void vec_antilogn_32x32(int32_t *  y, const int32_t*  x, int N);
void vec_antilog10_32x32(int32_t*  y, const int32_t*  x, int N);
void vec_antilog2_24x24 (f24 *  y, const f24*  x, int N);
void vec_antilogn_24x24 (f24 *  y, const f24*  x, int N);
void vec_antilog10_24x24(f24 *  y, const f24*  x, int N);
void vec_antilog2f (float32_t *  y, const float32_t*  x, int N);
void vec_antilognf (float32_t *  y, const float32_t*  x, int N);
void vec_antilog10f(float32_t *  y, const float32_t*  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`f24,int32_t, float32_t`	`x`	`N`	input data,Q6.25 or floating point
`int`	`N`		length of vectors
Output			
`f24,int32_t, float32_t`	`y`	`N`	output data,Q16.15 or floating point

Returned value

none

Restrictions

`x,y` – should not overlap

Conditions for optimum performance

`x,y` - aligned on 8-byte boundary
`N` - multiple of 2

Scalar versions

Prototypes

```
int32_t scl_antilog2_32x32 (int32_t x);
int32_t scl_antilogn_32x32 (int32_t x);
int32_t scl_antilog10_32x32(int32_t x);
f24 scl_antilog2_24x24 (f24 x);
f24 scl_antilogn_24x24 (f24 x);
f24 scl_antilog10_24x24(f24 x);
float32_t scl_antilog2f (float32_t x);
float32_t scl_antilognf (float32_t x);
float32_t scl_antilog10f(float32_t x);
```

Arguments

Type	Name	Description
Input		
`f24, int32_t, float32_t`	`x`	input data, Q6.25 or floating point

Returned value result, Q16.15 or floating point

2.3.5 Square Root

Description

These routines calculate square root.
NOTE: functions return 0x80000000 on negative argument for 32-bit outputs or 0x8000 for 16-bit outputs

Two versions of functions available: regular version (`vec_sqrt16x16, vec_sqrt24x24, vec_sqrt32x32, vec_sqrt64x32`) with arbitrary arguments and faster version (`vec_sqrt24x24_fast, vec_sqrt32x32_fast`) that apply some restrictions.

Precision

4 variants available:

Type	Description
16x16	16-bit inputs, 16-bit output. Accuracy: 2 LSB
24x24	24-bit inputs, 24-bit output. Accuracy: (2.6e-7*y+1LSB). Available for HiFi3/HiFi3z cores only
32x32	32-bit inputs, 32-bit output. Accuracy: (2.6e-7*y+1LSB)
64x32	64-bit input, 32-bit output. Accuracy: 2 LSB

Algorithm

$$y_n = \sqrt{x_n}$$

Prototype

```
void vec_sqrt16x16 (      int16_t*   y, const int16_t  *  x, int N);
void vec_sqrt24x24 (      f24 *  y, const f24 *  x, int N);
void vec_sqrt32x32 (      int32_t*  y, const int32_t*  x, int N);
void vec_sqrt64x32 (      int32_t*  y, const int64_t *  x, int N);
void vec_sqrt24x24_fast(  f24 *  y, const f24 *  x, int N);
void vec_sqrt32x32_fast(  int32_t*  y, const int32_t*  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int64_t, f24, int32_t, int16_t`	`x`	`N`	input data, Q63, Q31, Q15
`int`	`N`		length of vectors
Output			
`f24,int32_t, int16_t`	`y`	`N`	output data, Q31, Q15

Returned value none

Restrictions

Regular versions (`vec_sqrt16x16, vec_sqrt24x24, vec_sqrt32x32, vec_sqrt64x32`):
`x,y` – should not overlap

Faster versions (`vec_sqrt24x24_fast, vec_sqrt32x32_fast`):
`x,y` – should not overlap
`x,y` - aligned on 8-byte boundary
`N` - multiple of 2

Scalar versions

Prototypes

```
int16_t  scl_sqrt16x16(int16_t x);
f24      scl_sqrt24x24(f24 x);
int32_t  scl_sqrt32x32(int32_t x);
int32_t  scl_sqrt64x32(int64_t x);
```

Arguments

Type	Name	Description
Input		
`int64_t, f24, int32_t, int16_t`	`x`	input data, Q63, Q31, Q15

Returned value result, Q31, Q15

2.3.6 Reciprocal Square Root

Description

These routines return the fractional and exponential portion of the reciprocal square root of a vector `x` of Q31 or Q15 numbers. Since the reciprocal square root is always greater than 1, they return fractional portion `frac` in Q(31-`exp`) or Q(15-`exp`) format and exponent exp so true reciprocal value in the Q0.31/Q0.15 may be found by shifting fractional part left by exponent value.
NOTE: `scl_rsqrt32x32()` uses packed output for mantissa/exponent. To take a full precision, just call vectorized counterpart.
Mantissa accuracy is 1 LSB, so relative accuracy is:

`vec_rsqrt16x16, scl_rsqrt16x16`	6.2e-5
`scl_rsqrt32x32`	2.4e-7
`vec_rsqrt32x32`	9.2e-10

Precision

2 variants available:

Type	Description
32x32	32-bit input, 32-bit output.
16x16	16-bit input, 16-bit output.

Algorithm

$$frac_n \cdot 2^{exp_n} = 1/\sqrt{x_n}, n = \overline{0...N-1}$$

Prototype

```
void vec_rsqrt32x32 (
                  int32_t *  frac, int16_t *exp,
                  const int32_t *  x, int N)
void vec_rsqrt16x16 (
                  int16_t *  frac, int16_t *exp,
                  const int16_t *  x, int N)
```

Arguments

Type	Name	Size	Description
Input			
`int32_t,int16_t`	`x`	`N`	input data, Q31 or Q15
`int`	`N`		length of vectors
Output			
`int32_t, int16_t`	`frac`	`N`	fractional part of result, Q(31-`exp`) or Q(15-`exp`)
`int16_t`	`exp`	`N`	exponent of result

Returned value

None

Restrictions

`x,frac,exp` should not overlap

Scalar versions

Prototype

```
uint32_t scl_rsqrt32x32 (int32_t x)
uint32_t scl_rsqrt16x16 (int16_t x)
```

Arguments

Type	Name	Description
Input		
`int32_t, int16_t`	`x`	input data, Q31 or Q15

Returned value

packed value:
`scl_rsqrt32x32():`
bits 23…0 fractional part
bits 31…24 exponent
`scl_rsqrt16x16():`
bits 15…0 fractional part
bits 31…16 exponent

2.3.7 Sine/Cosine

Description

Fixed-point functions calculate `sin(pi*x)` or `cos(pi*x)` for numbers written in Q31 format. Return results in the same format. Floating point functions compute `sin(x)` or `cos(x)`.

Two versions of functions available: regular version (`vec_sine24x24, vec_cosine24x24, vec_sine32x32, vec_cosine32x32, vec_sinef, vec_cosinef`) with arbitrary arguments and faster version (`vec_sine24x24_fast, vec_cosine24x24_fast, vec_sine32x32_fast, vec_cosine32x32_fast`) that apply some restrictions.

NOTE:
1. Scalar floating point functions are compatible with standard ANSI C routines and set `errno` and exception flags accordingly.
2. Floating point functions limit the range of allowable input values: [-102940.0, 102940.0] Whenever the input value does not belong to this range, the result is set to `NaN`.

Precision

3 variants available:

Type	Description
24x24	24-bit inputs, 24-bit output. Accuracy: 74000(3.4e-5). Available for HiFi3/HiFi3z cores only
32x32	32-bit inputs, 32-bit output. Accuracy: 1700 (7.9e-7)
f	floating point. Accuracy 2 ULP. Requires VFPU core option

Algorithm

For fixed point:

$y_n = \sin(\pi x_n), n = \overline{0 \ldots N-1}$ or

$y_n = \cos(\pi x_n), n = \overline{0 \ldots N-1}$

For floating point:

$y_n = \sin(x_n), n = \overline{0 \ldots N-1}$ or

$y_n = \cos(x_n), n = \overline{0 \ldots N-1}$

Prototypes

```
void vec_sine24x24   (  f24 *  y,const f24 *  x, int N);
void vec_cosine24x24(f24 *  y, const f24 *  x, int N);
void vec_sine32x32   (  int32_t *  y, const int32_t *  x, int N);
void vec_cosine32x32(int32_t *  y, const int32_t *  x, int N);
void vec_sinef       (  float32_t *  y, const float32_t *  x, int N);
void vec_cosinef     (float32_t *  y, const float32_t *  x, int N);
void vec_sine24x24_fast  (f24 *  y, const f24 *  x, int N);
void vec_cosine24x24_fast(f24 *  y, const f24 *  x, int N);
void vec_sine32x32_fast  (int32_t *  y, const int32_t *  x, int N);
void vec_cosine32x32_fast(int32_t *  y, const int32_t *  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`f24, int32_t, float32_t`	`x`	`N`	input data,Q31 or floating point
`int`	`N`		length of vectors
Output			
`f24, int32_t, float32_t`	`y`	`N`	Result,Q31 or floating point

Returned value

None

Restrictions

Regular versions (`vec_sine24x24, vec_cosine24x24, vec_sine32x32, vec_cosine32x32, vec_sinef, vec_cosinef`):
`x,y` – should not overlap

Faster versions (`vec_sine24x24_fast, vec_cosine24x24_fast, vec_sine32x32_fast, vec_cosine32x32_fast`):
`x,y` – should not overlap
`x,y` - aligned on 8-byte boundary
`N` - multiple of 2

Scalar versions

Prototypes

```
f24 scl_sine24x24   (f24 x);
f24 scl_cosine24x24 (f24 x);
int32_t scl_sine32x32   (int32_t x);
int32_t scl_cosine32x32 (int32_t x);
float32_t scl_sinef   (float32_t x);
float32_t scl_cosinef (float32_t x);
```

Arguments

Type	Name	Description
Input		
`f24, int32_t, float32_t`	`x`	input data, Q31 or floating point

Returned value

result, Q31 or floating point

2.3.8 Tangent

Description

Fixed point functions calculate `tan(pi*x)` for number written in Q31. Floating point functions compute `tan(x)`.

NOTE:

1. Scalar floating point function is compatible with standard ANSI C routines and sets `errno` and exception flags accordingly.
2. Floating point functions limit the range of allowable input values: [-9099, 9099]. Whenever the input value does not belong to this range, the result is set to `NaN`.

Precision

3 variants available:

Type	Description
24x24	24-bit inputs, 32-bit outputs. Accuracy: (1.3e-4*`y`+1 LSB) if `abs(y)<=464873` (`14.19` in Q15) or `abs(x)<pi*0.4776`. Available for HiFi3/HiFi3z cores only
32x32	32-bit inputs, 32-bit outputs. Accuracy: (1.3e-4*`y`+1 LSB) if `abs(y)<=464873` (`14.19` in Q15) or `abs(x)<pi*0.4776`
f	floating point, Accuracy: 2 ULP. Requires VFPU core option

Algorithm

for fixed point:

$$y_n = \tan(\pi x_n), n = \overline{0...N-1}$$

for floating point

$$y_n = \tan(x_n), n = \overline{0...N-1}$$

Prototype

```
void vec_tan24x24 (int32_t*    y, const f24 * x, int N);
void vec_tan32x32 (int32_t*    y, const int32_t * x, int N);
void vec_tanf (float32_t * y, const float32_t * x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`f24, int32_t, float32_t`	`x`	`N`	input data, Q31 or floating point
`int`	`N`		length of vectors
Output			
`int32_t, float32_t`	`y`	`N`	result, Q16.15 or floating point

Returned value

none

Restrictions

`x,y` – should not overlap

Conditions for optimum performance

`x,y` - aligned on 8-byte boundary
`N` - multiple of 2

Scalar versions

Prototype

```
int32_t scl_tan24x24 (f24 x);
int32_t scl_tan32x32 (int32_t x);
float32_t scl_tanf (float32_t x);
```

Arguments

Type	Name	Description
Input		
f24, int32_t, float32_t	x	input data,Q31 or floating point

Returned value

result, Q16.15 or floating point

2.3.9 Arctangent

Description

Functions calculate arctangent of number. Fixed point functions scale down the output to `pi`

NOTE:
1. Scalar floating point function is compatible with standard ANSI C routines and sets `errno` and exception flags accordingly

Precision

3 variants available:

Type	Description
24x24	24-bit inputs, 24-bit output. Accuracy: 74000 (3.4e-5). Available for HiFi3/HiFi3z cores only
32x32	32-bit inputs, 32-bit output. Accuracy: 42 (2.0e-8)
f	floating point. Accuracy: 2 ULP. Requires VFPU core option

Algorithm

for fixed point

$$z_n = \arctan(x_n)/\pi, n = \overline{0...N-1}$$

for floating point

$$z_n = \arctan(x_n), n = \overline{0...N-1}$$

Prototype

```
void vec_atan24x24 (f24 *  z,
              const f24 *  x,
              int N );
void vec_atan32x32 (int32_t *  z,
              const int32_t *  x,
              int N );
void vec_atanf (    float32_t *  z,
              const float32_t *  x,
              int N );
```

Arguments

Type	Name	Size	Description
Input			
f24, int32_t, float32_t	x	N	input data, Q31 or floating point
int	N		length of vectors
Output			
f24, int32_t, float32_t	z	N	result, Q31 or floating point

Returned value

None

Restrictions `x,z` should not overlap

Conditions for optimum performance `x,z` aligned on 8-byte boundary
`N` multiple of 2

Scalar versions

Prototype

```
f24 scl_atan24x24 (f24 x);
int32_t scl_atan32x32 (int32_t x);
float32_t scl_atanf (float32_t x);
```

Arguments

Type	Name	Description
Input		
`f24, int32_t, float32_t`	`x`	input data, Q31 or floating point

Returned value result, Q31 or floating point

2.3.10 Full Quadrant Arctangent

Description The functions compute the full quadrant arc tangent of the ratio `y/x`. Floating point finctions is in radians. Fixed point functions scale its output by pi.
NOTE:

1. Scalar floating point function is compatible with standard ANSI C routines and sets `errno` and exception flags accordingly
2. Scalar floating point function assigns `EDOM` to `errno` whenever `y==0` and `x==0`.

Precision 2 variants available:

Type	Description
24x24	24-bit inputs, 24-bit output. Accuracy: 768 (3.57e-7). Available for HiFi3/HiFi3z cores only
f	floating point. Accuracy: 2 ULP. Requires VFPU core option

Algorithm

$$z_n = \arctan(y_n / x_n), n = \overline{0...N-1}$$

Prototype

```
void vec_atan2f (float32_t * z, const float32_t * y, const float32_t * x,int N);
void vec_atan2_24x24 (f24 * z, const f24 * y, const f24 * x,int N);
```

Arguments

Type	Name	Size	Description
Input			
`f24, float32_t`	`x`	`N`	input data, Q31 or floating point
`f24, float32_t`	`y`	`N`	input data, Q31 or floating point
`int`	`N`		length of vectors
Output			
`f24, float32_t`	`z`	`N`	result, Q31 or floating point

Returned value None

Restrictions `x, y, z` should not overlap

Scalar versions

Prototype

```
float32_t scl_atan2f (float32_t y, float32_t x);
f24 scl_atan2_24x24 (f24 y, f24 x);
```

Arguments

Type	Name	Description
Input		
`f24, float32_t`	`y`	input data, Q31 or floating point
`f24, float32_t`	`x`	input data, Q31 or floating point

Returned value result, Q31 or floating point

2.3.11 Hyperbolic Tangent

Description The functions compute the hyperbolic tangent of input argument. 32-bit fixed-point functions accept inputs in Q6.25 and form outputs in Q16.15 format.

Precision 1 variant available:

Type	Description
32x32	32-bit inputs, 32-bit output. Accuracy: 2 LSB

Algorithm

$$y_n = \tanh(x_n), n = \overline{0 \ldots N-1}$$

Prototype

```
void vec_tanh32x32 (int32_t * y, const int32_t * x,int N);
```

Arguments

Type	Name	Size	Description
Input			
`int32_t`	`x`	`N`	input data, Q6.25
`int`	`N`		length of vectors
Output			
`int32_t`	`y`	`N`	result, Q16.15

Returned value None

Restrictions `x, y` should not overlap

Scalar versions

Prototype

```
int32_t scl_tanh32x32 (int32_t x);
```

Arguments

Type	Name	Description
Input		
`int32_t`	`x`	input data, Q6.25

Returned value result, Q16.15

2.3.12 Sigmoid

Description The functions compute the sigmoid of input argument. 32-bit fixed-point functions accept inputs in Q6.25 and form outputs in Q16.15 format.

Precision 1 variant available:

Type	Description
32x32	32-bit inputs, 32-bit output. Accuracy: 2 LSB

Algorithm

$$y_n = \frac{1}{1+\exp(-x_n)}, n = \overline{0 \ldots N-1}$$

Prototype

```
void vec_sigmoid32x32 (int32_t * y, const int32_t * x,int N);
```

Arguments

Type	Name	Size	Description
Input			
`int32_t`	`x`	`N`	input data, Q6.25
`int`	`N`		length of vectors
Output			
`int32_t`	`y`	`N`	result, Q16.15

Returned value None

Restrictions `x, y` should not overlap

Scalar versions

Prototype

```
int32_t scl_sigmoid32x32 (int32_t x);
```

Arguments

Type	Name	Description
Input		
int32_t	x	input data, Q6.25

Returned value result, Q16.15

2.3.13 Softmax

Description The function computes the softmax (normalized exponential function) of input data. 32-bit fixed-point functions accept inputs in Q6.25 and form outputs in Q16.15 format.

Precision 1 variant available:

Type	Description
32x32	32-bit inputs, 32-bit output. Accuracy: 2 LSB (see Note below)

Note: Accuracy of function may depend on amount of data and their distribution. Given accuracy is achieved for N=2 for any pair of data from input domain.

Algorithm

$$y_n = \frac{\exp(x_n)}{\sum_k \exp(x_k)}, n = \overline{0...N-1}$$

Prototype

```
void vec_softmax32x32 (int32_t * y, const int32_t * x,int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t	x	N	input data, Q6.25
int	N		length of vectors
Output			
int32_t	y	N	result, Q16.15

Returned value None

Restrictions `x, y` should not overlap

2.3.14 Integer to Float Conversion

Description Routine converts integer to float and scales result up by `2^t`.

Precision 1 variant available:

Type	Description
f	32-bit input, floating point output. Requires VFPU core option

Algorithm

$$y_n = x_n \cdot 2^t, n = \overline{0...N-1}$$

Prototype

```
void   vec_int2float
      ( float32_t  *  y,
        const int32_t  *  x,
        int t, int N);
```

Arguments

Type	Name	Size	Description
Input			
int32_t	x	N	input data, integer
int	t		scale factor

`int`	`N`		length of vectors
Output			
`float32_t`	`y`	`N`	Conversion result, floating point

Returned value None

Restrictions `t` should be in range `-126...126`

Scalar version

Prototype

```
float32_t scl_int2float (int32_t x, int t);
```

Arguments

Type	Name	Description
Input		
`int32_t`	`x`	input data, integer

Returned value result, floating point

Restrictions `t` should be in range `-126...126`

2.3.15 Float to Integer Conversion

Description Routine scales floating point input down by `2^t` and converts it to integer with saturation

Precision 1 variant available:

Type	Description
f	floating point input, 32-bit output. Requires VFPU core option

Algorithm

$$y_n = x_n \cdot 2^{-t}, n = \overline{0...N-1}$$

Prototype

```
void   vec_float2int
         (  int32_t *  y,
            const float32_t *  x,
            int t, int N);
```

Arguments

Type	Name	Size	Description
Input			
`float32_t`	`x`	`N`	input data, floating point
`int`	`t`		scale factor
`int`	`N`		length of vectors
Output			
`int32_t`	`y`	`N`	Conversion results, integers

Returned value None

Restrictions `t` should be in range `-126...126`

Scalar version

Prototype

```
int32_t scl_float2int (float32_t x, int t);
```

Arguments

Type	Name	Description
Input		
`float32_t`	`x`	input data, floating point

Returned value result, integer

Restrictions `t` should be in range `-126...126`

2.4 Complex Mathematics

2.4.1 Complex Magnitude

Description Routines compute complex magnitude or its reciprocal

Precision 3 variants available:

Type	Description
f	floating point input, 32-bit output. Requires VFPU core option

Algorithm $y_n = abs(x_n), n = \overline{0 \ldots N-1}$

Prototype

```
void vec_complex2mag   (float32_t  *  y, const complex_float  *  x, int N);
void vec_complex2mag16x16 (int16_t* z, complex16_t* x, int N);
void vec_complex2mag32x32 (int32_t* z, complex32_t* x, int N)
```

Arguments

Type	Name	Size	Description
Input			
complex_float complex32_t complex16_t	x	N	input data
int	N		length of vectors
Output			
float32_t int32_t int16_t	y z z	N	magnitude

Returned value None

Restrictions x should be 8 byte aligned

Scalar version

Prototype

```
float32_t  scl_complex2mag    (complex_float x);
float32_t  scl_complex2invmag (complex_float x);
```

Arguments

Type	Name	Description
Input		
complex_float	x	input data

Returned value result, floating point

Restrictions None

2.4.2 Complex Inverse Magnitude

Description Routines compute complex magnitude or its reciprocal

Precision 1 variant available:

Type	Description
f	floating point input, 32-bit output. Requires VFPU core option

Algorithm $y_n = 1/abs(x_n), n = \overline{0 \ldots N-1}$

Prototype

```
void vec_complex2invmag (float32_t  *  y, const complex_float  *  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
complex_float	x	N	input data
int	N		length of vectors
Output			

`float32_t`	`y`	`N`	Reciprocal of magnitude

Returned value None

Restrictions x should be 8 byte aligned

Scalar version

Prototype

```
float32_t  scl_complex2mag    (complex_float x);
float32_t  scl_complex2invmag (complex_float x);
```

Arguments

Type	Name	Description
Input		
`complex_float`	`x`	input data

Returned value result, floating point

Restrictions None

2.4.3 Complex to Complex Multiplication

Description This routine does element wise complex to complex multiplication of two complex valued vectors. NOTE: function returns zero if `N` is less or equal to zero

Precision 3 variants available:

Type	Description
32x32	32-bit data, 32-bit output
16x16	16-bit data, 16-bit output
f	floating point. Requires SP-VFPU/SFPU core option

Algorithm $z_n = (x_n * \ y_n),\ n = \overline{0 \ldots N-1}$

Prototype

```
void vec_cplx2cplx_mult32x32 (complex32_t*  z, complex32_t*  x, complex32_t*  y,
int N);
void vec_cplx2cplx_mult16x16 (complex16_t*  z, complex16_t*  x, complex16_t*  y,
int N);
void vec_cplx2cplx_multf (complex_float*  z, complex_float*  x, complex_float*
y, int N);
```

Arguments

Type	Name	Size	Description
Input			
`complex32_t, complex16_t, complex_float`	`x, y`	`N`	input data
`int`	`N`		length of vector
`complex32_t, complex16_t, complex_float`	`z`	`N`	output data

Returned value None

Restrictions `x,y,z` aligned on 8-byte boundary

2.4.4 Complex Vector to Real Vector Multiplication

Description This routine does element wise complex to real multiplication, of one complex valued vector with another real valued vector.

Precision 3 variants available:

Type	Description

32x32	32-bit data, 32-bit output
16x16	16-bit data, 16-bit output
f	floating point. Requires SP-VFPU/SFPU core option

Algorithm $z_n = (x_n * \ y_n), \ n = \overline{0 \ldots N-1}$

Prototype

```
void vec_cplx2real_multv32x32 (complex32_t*  z, complex32_t*  x, int32_t*  y,
int N);
void vec_cplx2real_multv16x16 (complex16_t*  z, complex16_t*  x, int16_t*  y,
int N);
void vec_cplx2real_multvf (complex_float*  z, complex_float*  x, float32_t*  y,
int N);
```

Arguments

Type	Name	Size	Description
Input			
complex32_t, complex16_t, complex_float	x	N	input data
int32_t, int16_t, float32_t	y	1	input data
int	N		length of vector
complex32_t, complex16_t, complex_float	Z	N	output data

Returned value None

Restrictions x,y aligned on 8-byte boundary

2.4.5 Complex Vector to Real Scalar Multiplication

Description This routine does element wise complex to real multiplication, of a complex valued vector with a real valued scalar.

Precision 3 variants available:

Type	Description
32x32	32-bit data, 32-bit output
16x16	16-bit data, 16-bit output
f	floating point. Requires SP-VFPU/SFPU core option

Algorithm $z_n = (x_n * \ y \ ), \ n = \overline{0 \ldots N-1}$

Prototype

```
void vec_cplx2real_mults32x32 (complex32_t*  z, complex32_t*  x, int32_t   y,
int N);
void vec_cplx2real_mults16x16 (complex16_t*  z, complex16_t*  x, int16_t   y,
int N);
void vec_cplx2real_multsf (complex_float*  z, complex_float*  x, float32_t   y,
int N);
```

Arguments

Type	Name	Size	Description
Input			
complex32_t, complex16_t, complex_float	x	N	input data
int32_t, int16_t, float32_t	y	N	input data
int	N		length of vector
complex32_t, complex16_t, complex_float	Z	N	output data

Returned value None

Restrictions `x,y` aligned on 8-byte boundary

2.4.6 Complex Conjugate

Description This routine does element wise conjugate of a complex valued vector.

Precision 3 variants available:

Type	Description
32x32	32-bit data, 32-bit output
16x16	16-bit data, 16-bit output
f	floating point. Requires SP-VFPU/SFPU core option

Algorithm $z_n = conj(x_n),\ n = \overline{0 \dots N-1}$

Prototype

```
void vec_cplxconj32x32(complex32_t*  z, const complex32_t*  x, int N);
void vec_cplxconj16x16(complex16_t*  z, const complex16_t*  x, int N);
void vec_cplx_Conjf(complex_float* z, const complex_float*  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`complex32_t, complex16_t, complex_float`	`x`	`N`	input data
`int`	`N`		length of vector
`complex32_t, complex16_t, complex_float`	`z`	`N`	output data

Returned value None

Restrictions `x` aligned on 8-byte boundary

2.5 Vector Operations

2.5.1 Vector Dot Product

Description

These routines take two vectors and calculates their dot product. Two versions of routines are available: regular versions (`vec_dot24x24, vec_dot32x16, vec_dot32x32, vec_dot16x16, vec_dotf`) work with arbitrary arguments, faster versions (`vec_dot24x24_fast, vec_dot32x16_fast, vec_dot32x32_fast, vec_dot16x16_fast`) apply some restrictions.

Precision

8 variants available:

Type	Description
64x32	64x32-bit data, 64-bit output (fractional multiply Q63xQ31->Q63)
64x64	64x64-bit data, 64-bit output (fractional multiply Q63xQ63->Q63)
64x64i	64x64-bit data, 64-bit output (low 64 bit of integer multiply)
24x24	24x24-bit data, 64-bit output. Available for HiFi3/HiFi3z cores only
32x16	32x16-bit data, 64-bit output
32x32	32x32-bit data, 64-bit output
16x16	16x16-bit data, 64-bit output for regular version and 32-bit for fast version
f	floating point. Requires VFPU core option

Algorithm

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

Prototype

```
int64_t vec_dot64x32 (const int64_t   *  x,const int32_t * y, int N);
int64_t vec_dot64x64 (const int64_t   *  x,const int64_t * y, int N);
int64_t vec_dot64x64i(const int64_t   *  x,const int64_t * y, int N);
int64_t vec_dot24x24 (const f24       *x, const f24      * y, int N);
int64_t vec_dot32x16 (const int32_t   *x, const int16_t  * y, int N);
int64_t vec_dot16x16 (const int16_t   *x, const int16_t  * y, int N);
int64_t vec_dot32x32 (const int32_t   *x, const int32_t  * y, int N);
float32_t vec_dotf   (const float32_t *x, const float32_t* y, int N);

int64_t vec_dot64x32_fast (const int64_t * x,const int32_t * y,int N);
int64_t vec_dot64x64_fast (const int64_t * x,const int64_t * y,int N);
int64_t vec_dot64x64i_fast(const int64_t * x,const int64_t * y,int N);
int64_t vec_dot24x24_fast (const f24     *  x, const f24     *  y, int N);
int64_t vec_dot32x16_fast (const int32_t *  x, const int16_t *  y, int N);
int64_t vec_dot32x32_fast (const int32_t *  x, const int32_t *  y, int N);
int32_t vec_dot16x16_fast (const int16_t *  x, const int16_t *  y, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int64_t, f24, int32_t, int16_t, float32_t`	`x`	`N`	input data, Q63, Q31, Q15 or floating point
`int64_t, f24,int16_t, float32_t`	`y`	`N`	input data, Q63, Q31, Q15 or floating point
`int`	`N`		length of vectors

Returned value

dot product of all data pairs, Q63, Q31 or floating point

Restrictions

Regular versions (`vec_dot64x32, vec_dot64x64, vec_dot64x64i, vec_dot24x24, vec_dot32x16, vec_dot32x32,vec_dot16x16, vec_dotf`):
None

Faster versions (`vec_dot64x32_fast, vec_dot64x64_fast, vec_dot64x64i_fast, vec_dot24x24_fast, vec_dot32x16_fast, vec_dot32x32_fast, vec_dot16x16_fast`):
`x,y` - aligned on 8-byte boundary

`N` - multiple of 4

`vec_dot16x16_fast` utilizes 32-bit saturating accumulator, so, input data should be scaled properly to avoid erroneous results especially in case of heterogenic data.

2.5.2 Vector Sum

Description

This routine makes pair wise saturated summation of vectors. Two versions of routines are available: regular versions (`vec_add32x32, vec_add24x24, vec_add16x16, vec_addf`) work with arbitrary arguments, faster versions (`vec_add32x32_fast, vec_add24x24_fast, vec_add16x16_fast`) apply some restrictions.

Precision

4 variants available:

Type	Description
32x32	32-bit inputs, 32-bit output
24x24	24-bit inputs, 24-bit output. Available for HiFi3/HiFi3z cores only
16x16	16-bit inputs, 16-bit output
f	floating point. Requires VFPU core option

Algorithm

$$z_n = x_n + y_n, n = \overline{0...N-1}$$

Prototype

```
void vec_add32x32 ( int32_t*  z, const int32_t*  x, const int32_t*  y, int N);
void vec_add24x24 ( f24 *  z, const f24 *  x, const f24 *  y, int N);
void vec_add16x16 ( int16_t*  z, const int16_t*  x, const int16_t*  y, int N);
void vec_addf(float32_t*  z, const float32_t*  x, const float32_t*  y, int N);

void vec_add32x32_fast(int32_t*  z, const int32_t*  x, const int32_t*  y,int N);
void vec_add24x24_fast(f24 *  z, const f24 *  x, const f24 *  y, int N);
void vec_add16x16_fast(int16_t*  z, const int16_t*  x, const int16_t*  y,int N);
```

Arguments

Type	Name	Size	Description
Input			
`f24, int32_t, int16_t or float32_t`	`x`	`N`	input data
`f24, int32_t, int16_t or float32_t`	`y`	`N`	input data
`int`	`N`		length of vectors
Output			
`f24, int32_t, int16_t or float32_t`	`z`	`N`	output data

Returned value

none

Restrictions

Regular versions (`vec_add32x32, vec_add24x24, vec_add16x16, vec_addf`):
`x,y,z` - should not be overlapped

Faster versions (`vec_add32x32_fast, vec_add24x24_fast, vec_add16x16_fast`):
`z,x,y` - aligned on 8-byte boundary
`N` - multiple of 4

2.5.3 Power of a Vector

Description

These routines compute power of vector with scaling output result by `rsh` bits. Fixed point routines make accumulation in the 64-bit wide accumulator and output may scaled down with saturation by `rsh` bits. So, if representation of `x` input is `Qx`, result will be represented in `Q(2x-rsh)` format.

Two versions of routines are available: regular versions (`vec_power24x24, vec_power32x32, vec_power16x16, vec_powerf`) work with arbitrary arguments, faster versions (`vec_power24x24_fast, vec_power32x32_fast, vec_power16x16_fast`) apply some restrictions.

Precision

4 variants available:

Type	Description
24x24	24x24-bit data, 64-bit output. Available for HiFi3/HiFi3z cores only
32x32	32x32-bit data, 64-bit output
16x16	16x16-bit data, 64-bit output
f	floating point. Requires VFPU core option

Algorithm

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

Prototype

```
int64_t     vec_power24x24 ( const f24 *  x,
                             int rsh, int N);
int64_t     vec_power32x32 ( const int32_t *  x,
                             int rsh, int N);
int64_t     vec_power16x16 ( const int16_t *  x,
                             int rsh, int N);
float32_t   vec_powerf     ( const float32_t *  x, int N);

int64_t     vec_power24x24_fast ( const f24 *  x,
                             int rsh, int N)
int64_t     vec_power32x32_fast ( const int32_t *  x,
                             int rsh, int N)
int64_t     vec_power16x16_fast ( const int16_t *  x,
                             int rsh, int N)
```

Arguments

Type	Name	Size	Description
Input			
`f24, int32_t, int16_t, float32_t`	`x`	`N`	input data, Q31, Q15 or floating point
`int`	`rsh`		right shift of result (only for fixed point routines): for `vec_power32x32()`: `rsh` should be in range 31...62 for `vec_power24x24()`: `rsh` should be in range 15...46 for `vec_power16x16()`: `rsh` should be in range 0...31
`int`	`N`		length of vector

Returned value

Sum of squares of a vector, Q(`2x-rsh`) or floating point

Restrictions

For regular versions (`vec_power24x24, vec_power32x32, vec_power16x16, vec_powerf`): none

For faster versions (`vec_power24x24_fast, vec_power32x32_fast, vec_power16x16_fast`)
`x` - aligned on 8-byte boundary
`N` - multiple of 4

2.5.4 Vector Scaling with Saturation

Description

These routines make shift with saturation of data values in the vector by given scale factor (degree of 2). 24-bit routine works with f24 data type and faster while 32-bit version keep all 32-bits and slower. Functions `vec_scale()` make multiplication of vector to coefficient which is not a power of 2.
Two versions of routines are available: regular versions (`vec_shift24x24, vec_shift32x32, vec_shift16x16, vec_shiftf, vec_scale32x24, vec_scale32x32, vec_scale24x24, vec_scale16x16, vec_scalef, vec_scale_sf`) work with arbitrary arguments, faster versions (`vec_shift24x24_fast, vec_shift32x32_fast, vec_shift16x16_fast,`

vec_scale32x24_fast, vec_scale24x24_fast, vec_scale32x32_fast, vec_scale16x16_fast) apply some restrictions.
For floating point:
Function vec_shiftf() makes scaling without saturation of data values in the vector by given scale factor (degree of 2). Functions vec_scalef() and vec_scale_sf() make multiplication of input vector to coefficient which is not a power of 2. vec_scalef() makes scaling without saturations, vec_scale_sf() allows to saturate results on given boundaries.

Precision

5 variants available:

Type	Description
24x24	24-bit input, 24-bit output. Available for HiFi3/HiFi3z cores only
32x24	32-bit input, 32-bit output, 24-bit scale factor. Available for HiFi3/HiFi3z cores only
32x32	32-bit input, 32-bit output
16x16	16-bit input, 16-bit output
f	floating point. Requires VFPU core option

Algorithm

$$r_n = x_n \cdot 2^t$$

Prototype

```
void vec_shift24x24 (     f24 *  y,
                    const f24 *  x, int t, int N);
void vec_shift32x32 (     int32_t *  y,
                    const int32_t *  x, int t, int N);
void vec_shift16x16 (     int16_t *  y,
                    const int16_t *  x, int t, int N);
void vec_shiftf     (     float32_t *  y,
                    const float32 *  x, int t, int N);
void vec_shift24x24_fast (     f24 *  y,
                    const f24 *  x, int t, int N);
void vec_shift32x32_fast (     int32_t *  y,
                    const int32_t *  x, int t, int N);
void vec_shift16x16_fast (     int16_t *  y,
                    const int16_t *  x, int t, int N);
```

Arguments

Type	Name	Size	Description
Input			
f24, int32_t, int16_t or float32_t	x	N	input data, Q31, Q15 or floating point
int	t		shift count. If positive, it shifts left with saturation, if negative it shifts right
int	N		length of vector
Output			
f24, int32_t, int16_t or float32_t	y	N	output data, Q31, Q15 or floating point

Prototype

non-power 2 scaling

```
void vec_scale32x24 (     int32_t *  y,
                    const int32_t *  x,
                    f24 s, int N);
void vec_scale32x32 (     int32_t *  y,
                    const int32_t *  x, int32_t s, int N);
void vec_scale24x24 (     f24 *  y,
                    const f24 *  x, f24 s, int N);
void vec_scale16x16 (     int16_t *  y,
                    const int16_t *  x, int16_t s, int N);
void vec_scalef     (     float32_t *  y,
                    const float32_t *  x, float32_t s, int N);
void vec_scale_sf   (     float32_t * restrict y,
                    const float32_t * restrict x,
                    float32_t s, float32_t fmin, float32_t fmax, int N);
void vec_scale32x24_fast (     int32_t *  y,
                    const int32_t *  x, f24 s, int N);
void vec_scale24x24_fast (     f24 *  y,
                    const f24 *  x, f24 s, int N);
void vec_scale32x32_fast (     int32_t*  y,
                    const int32_t *  x, int32_t s, int N);
void vec_scale16x16_fast (     int16_t *  y,
                    const int16_t *  x, int16_t s, int N);
```

Arguments

Type	Name	Size	Description
Input			
`f24, int32_t, int16_t or float32_t`	`x`	`N`	input data, Q31, Q15 or floating point
`f24, int16_t, float32_t`	`s`		scale factor, Q31, Q15 or floating point
`int`	`N`		length of vector
`float32_t`	`fmin`		lower bound of resulted values (for `vec_scale_sf()` only)
`float32_t`	`fmax`		upper bound of resulted values (for `vec_scale_sf()` only)
Output			
`f24, int32_t, int16_t or float32_t`	`y`	`N`	output data, Q31, Q15 or floating point

Returned value

None

Restrictions

For regular versions (`vec_shift24x24, vec_shift32x32, vec_shift16x16, vec_shiftf, vec_scale32x24, vec_scale32x32, vec_scale24x24, vec_scale16x16, vec_scalef, vec_scalesf`):
`x,y` should not overlap
`t` should be in range -31...31 for fixed-point functions and -129...146 for floating point

For faster versions (`vec_shift24x24_fast, vec_shift32x32_fast, vec_shift16x16_fast, vec_scale32x24_fast, vec_scale24x24_fast, vec_scale32x32_fast, vec_scale16x16_fast`):
`x,y` should not overlap
`t` should be in range -31...31
`x,y` - aligned on 8-byte boundary
`N` - multiple of 4

2.5.5 Common Exponent

Description

These functions determine the number of redundant sign bits for each value (as if it was loaded in a 32-bit register) and returns the minimum number over the whole vector. This may be useful for a FFT implementation to normalize data.
Floating point function returns `0-floor(log2(max(abs(x))))`. Returned result will be always in range `[-129...146]`.

Special cases

Input	Result
0	0
+/-Inf	-129
NaN	0

24-bit version is approximately 1.5 times faster but does not use lower 8 bits of numbers. 32-bit version use all 32-bits and delivers better dynamic range.
NOTES:
Faster versions of functions make the same task but in a different manner – they compute exponent of maximum absolute value in the array. It allows faster computations but not bitexact results – if minimum value in the array will be -2^n , fast function returns `max(0,30-n)` while non-fast function returns `(31-n)`.
Functions return zero if `N<=0`

Precision

4 variants available:

Type	Description
32	32-bit inputs
24	24-bit inputs. Available for HiFi3/HiFi3z cores only
16	16-bit inputs
f	floating point inputs. Requires VFPU core option

Algorithm

$$z_n = \min_{n=0\ldots N-1}\left(norm(x_n) \right)$$ non-fast version

$$z_n = \min_{n=0\ldots N-1}\left(norm(abs(x_n)) \right)$$ fast version

$$z_n = -floor\left(\log_2(\max_{n=0\ldots N-1}(abs(x_n))) \right)$$ for floating point

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

Prototype

```
int vec_bexp32 (const int32_t *  x, int N);
int vec_bexp24 (const f24     *  x, int N);
int vec_bexp16 (const int16_t *  x, int N);
int vec_bexpf  (const float32_t *  x, int N);

int vec_bexp32_fast (const int32_t *  x, int N);
int vec_bexp24_fast (const f24     *  x, int N);
int vec_bexp16_fast (const int16_t *  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`f24, int32_t, int16_t, float32_t`	`x`	`N`	input data
`int`	`N`		length of vector

Returned value

minimum exponent

Restrictions

non-fast functions (`vec_bexp16, vec_bexp24, vec_bexp32, vec_bexpf`):
none
for fast functions (`vec_bexp16_fast, vec_bexp24x24_fast, vec_bexp32x32_fast`):
`x,y` - aligned on 8-byte boundary
`N` - multiple of 4

Scalar versions

Prototype

```
int scl_bexp32 (int32_t x);
int scl_bexp24 (f24 x);
```

```
int scl_bexp16 (int16_t x);
int scl_bexpf  (float32_t x);
```

Arguments

Type	Name	Description
Input		
`f24, int32_t, int16_t, float32_t`	`x`	input data

Returned value

result

2.5.6 Elementwise Absolute of Vector

Description

This routine finds element wise absolute value of real data in a vector.
NOTE: function returns zero if `N` is less or equal to zero

Precision

3 variants available:

Type	Description
32x32	32-bit data, 32-bit output
16x16	16-bit data, 16-bit output
f	floating point. Requires SP-VFPU core option

Algorithm

$z_n = \text{abs}(x_n), n = \overline{0 \ldots N\text{-}1}$

Prototype

void vec_eleabs32x32 (const int32_t* x, int32_t* z, int N);
void vec_eleabs16x16 (const int16_t* x, int16_t* z, int N);
void vec_eleabsf (const float32_t* x, float32_t* z, int N);

Arguments

Type	Name	Size	Description
Input			
`int32_t, int16_t, float32_t`	`x`	`N`	input data
`int32_t, int16_t, float32_t`	`Z`	`N`	Output data
`int`	`N`		length of vector

Returned value

None

Restrictions

`x` aligned on 8-byte boundary

2.5.7 Vector Min/Max

Description

These routines find maximum/minimum value in a vector.
Two versions of functions available: regular version (`vec_min32x32, vec_max32x32, vec_min24x24, vec_max24x24,vec_max16x16, vec_min16x16,vec_maxf, vec_minf`) with arbitrary arguments and faster version (`vec_min32x32_fast, vec_max32x32_fast, vec_min24x24_fast, vec_max24x24_fast,vec_min16x16_fast, vec_min16x16_fast`) that apply some restrictions
NOTE: functions return zero if `N` is less or equal to zero

Precision

4 variants available:

Type	Description

32x32	32-bit data, 32-bit output
24x24	24-bit data, 24-bit output. Available for HiFi3/HiFi3z cores only
16x16	16-bit data, 16-bit output
f	floating point. Requires VFPU core option

Algorithm

$v = \min(x_n), n = \overline{0 \ldots N-1}$

or

$v = \max(x_n), n = \overline{0 \ldots N-1}$

Prototype

```
int32_t   vec_min32x32 (const int32_t  *  x, int N);
f24       vec_min24x24 (const f24      *  x, int N);
int16_t   vec_min16x16 (const int16_t  *  x, int N);
float32_t vec_minf     (const float32_t*  x, int N);
int32_t vec_max32x32   (const int32_t  *  x, int N);
f24     vec_max24x24   (const f24      *  x, int N);
int16_t vec_max16x16   (const int16_t  *  x, int N);
float32_t vec_maxf     (const float32_t*  x, int N);
int32_t vec_min32x32_fast (const int32_t*  x, int N);
f24     vec_min24x24_fast (const f24    *  x, int N);
int16_t vec_min16x16_fast (const int16_t*  x, int N);
int32_t vec_max32x32_fast (const int32_t*  x, int N);
f24     vec_max24x24_fast (const f24    *  x, int N);
int16_t vec_max16x16_fast (const int16_t*  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
f24,int32_t, int16_t, float32_t	x	N	input data
int	N		length of vector

Returned value

minimum or maximum value

Restrictions

For regular routines (vec_min32x32, vec_max32x32, vec_min24x24, vec_max24x24,vec_max16x16, vec_min16x16,vec_maxf, vec_minf) :
none
For faster routines (vec_min32x32_fast, vec_max32x32_fast, vec_min24x24_fast, vec_max24x24_fast,vec_min16x16_fast, vec_min16x16_fast):
x aligned on 8-byte boundary
N - multiple of 4

2.5.8 Elementwise Vector Subtraction

Description

This routine does pair wise subtraction of values in two vectors.
NOTE: function returns zero if N is less or equal to zero

Precision

3 variants available:

Type	Description
32x32	32-bit data, 32-bit output
16x16	16-bit data, 16-bit output
f	floating point. Requires SP-VFPU core option

Algorithm

$z_n = (x_n - y_n), n = \overline{0 \ldots N-1}$

Prototype

```
void vec_elesubf (float32_t*  z, float32_t*  x, float32_t*  y, int N);
void vec_elesub32x32 (int32_t*  z, int32_t*  x, int32_t*  y, int N);
void vec_elesub16x16 (int16_t*  z, int16_t*  x, int16_t*  y, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int32_t`, `int16_t`, `float32_t`	`x`	`N`	input data
`int`	`N`		length of vector

Returned value None

Restrictions `x, y` aligned on 8-byte boundary

2.5.9 Elementwise Vector Multiplication

Description This routine does pair wise multiplication of values in two vectors.
NOTE: function returns zero if `N` is less or equal to zero

Precision 3 variants available:

Type	Description
32x32	32-bit data, 32-bit output
16x16	16-bit data, 16-bit output
f	floating point. Requires SP-VFPU core option

Algorithm $z_n = (x_n * y_n), n = \overline{0 \ldots N-1}$

Prototype

```
void vec_elemult32x32(int32_t* z, int32_t* x, int32_t* y, int N);
void vec_elemult16x16 (int16_t* z, int16_t* x, int16_t* y, int N);
void vec_elemultf(float32_t* z, float32_t* x, float32_t* y, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int32_t`, `int16_t`, `float32_t`	`x`	`N`	input data
`int`	`N`		length of vector

Returned value None

Restrictions `x, y` aligned on 8-byte boundary

2.5.10 Vector Sum

Description This routine calculates cumulative sum of all elements in a real valued vector.
NOTE: function returns zero if `N` is less or equal to zero

Precision 3 variants available:

Type	Description
32x32	32-bit data, 32-bit output
16x16	16-bit data, 16-bit output
f	floating point. Requires SP-VFPU core option

Algorithm $z_n = sum(x_n), n = \overline{0 \ldots N-1}$

Prototype

```
float32_t vec_sumf (const float32_t *  x, int N);
int32_t vec_sum32x32 (const int32_t*  x, int N);
int16_t vec_sum16x16 (const int16_t*  x, int N);
```

Arguments

Type	Name	Size	Description
Input			

`int32_t,`
`int16_t,`
`float32_t`
	`x`	`N`	input data

`int`	`N`		length of vector

Returned value

vector sum

Restrictions

`x` aligned on 8-byte boundary
The sequential addition version has accuracy of 2 ULP.
The optimized version is faster than the sequential addition version, but the accuracy is 1800 ULP. The accuracy difference is because of the addition sequence.

2.5.11 Vector Mean

Description

This routine calculates mean value of elements in a real valued vector.
NOTE: function returns zero if `N` is less or equal to zero

Precision

3 variants available:

Type	Description
32x32	32-bit data, 32-bit output
16x16	16-bit data, 16-bit output
f	floating point. Requires SP-VFPU core option

Algorithm

$$z_n = \frac{\sum(x_n)}{N}, n = \overline{0 \ldots N-1}$$

Prototype

```
int32_t       vec_mean32x32 (const int32_t*  x, int N);
int16_t       vec_mean16x16 (const int16_t*  x, int N);
float32_t     vec_meanf (const float32_t *  x, int N);
```

Arguments

Type	Name	Size	Description
Input			

`int32_t,`
`int16_t,`
`float32_t`
	`x`	`N`	input data

`int`	`N`		length of vector

Returned value

minimum or maximum value

Restrictions

`x` aligned on 8-byte boundary
The sequential addition version has accuracy of 2 ULP.
The optimized version is faster than the sequential addition version, but the accuracy is 1800 ULP. The accuracy difference is because of the addition sequence.

2.5.12 Vector RMS

Description

This routine calculates root mean squared value of a real valued vector.
NOTE: function returns zero if `N` is less or equal to zero

Precision

3 variants available:

Type	Description
32x32	32-bit data, 32-bit output
16x16	16-bit data, 16-bit output
f	floating point. Requires SP-VFPU core option

Algorithm

$$z_n = \sqrt{\frac{\Sigma(x_n * x_n)}{N}}, n = \overline{0 \ldots N-1}$$

Prototype

```
int32_t       vec_rms32x32(const int32_t*  x, int N);
int16_t       vec_rms16x16(const int16_t*  x, int N);
float32_t     vec_rmsf (const float32_t *  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int32_t`, `int16_t`, `float32_t`	`x`	`N`	input data
`int`	`N`		length of vector

Returned value

rms value

Restrictions

`x` aligned on 8-byte boundary
The sequential operation version has accuracy of 2 ULP.
The optimized version is faster than the sequential operation version, but the accuracy is 18 ULP. The accuracy difference is because of the addition sequence.

2.5.13 Vector Variance

Description

This routine calculates variance of a real valued vector.
NOTE: function returns zero if `N` is less or equal to one.

Precision

3 variants available:

Type	Description
32x32	32-bit data, 32-bit output
16x16	16-bit data, 16-bit output
f	floating point. Requires SP-VFPU core option

Algorithm

$$z_n = \Sigma\frac{(x_i-\mu)^2}{N-1}, n = \overline{0 \ldots N-1}$$

Prototype

```
int32_t       vec_var32x32 (const int32_t*  x, int N);
int16_t       vec_var16x16 (const int16_t*  x, int N);
float32_t     vec_varf (const float32_t *  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int32_t`, `int16_t`, `float32_t`	`x`	`N`	input data
`int`	`N`		length of vector

Returned value

variance value

Restrictions

`x` aligned on 8-byte boundary
The sequential operation version has accuracy of 2 ULP.
The optimized version is faster than the sequential version, but the accuracy is 90 ULP. The accuracy difference is because of the addition sequence.

2.5.14 Vector Standard Deviation

Description

This routine calculates standard deviation of a real valued vector.

NOTE: function returns zero if `N` is less or equal to zero

Precision

3 variants available:

Type	Description
32x32	32-bit data, 32-bit output
16x16	16-bit data, 16-bit output
f	floating point. Requires SP-VFPU/SFPU core option

Algorithm

$sd = \sqrt{v},\ v = variance\ of\ x$

Prototype

```
int32_t       vec_stddev32x32 (const int32_t*  x, int N);
int16_t       vec_stddev16x16 (const int16_t*  x, int N);
float32_t     vec_stddevf (const float32_t *  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int32_t, int16_t, float32_t`	`x`	`N`	input data
`int`	`N`		length of vector

Returned value

standard deviation value

Restrictions

`x` aligned on 8-byte boundary
The sequential operation version has accuracy of 2 ULP.
The optimized version is faster than the sequential version, but the accuracy is 40 ULP. The accuracy difference is because of the addition sequence.

2.6 Matrix Operations

2.6.1 Matrix Multiply

Description

These functions compute the expression `z = 2^lsh * x * y` for the matrices `x` and `y`. The column 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`.
NOTE: `lsh` factor is not relevant for floating point routines.

Functions require scratch memory for storing intermediate data. This scratch memory area should be aligned on 8 byte boundary and its size is calculated by macros `SCRATCH_MTX_MPY24X24(M,N,P)`, `SCRATCH_MTX_MPY32X32(M,N,P)`, `SCRATCH_MTX,_MPY16X16(M,N,P)`

Two versions of functions available: regular version (`mtx_mpy24x24`, `mtx_mpy32x32`, `mtx_mpy16x16`, `mtx_mpyf`) with arbitrary arguments and faster version (`mtx_mpy24x24_fast`, `mtx_mpy32x32_fast`, `mtx_mpy16x16_fast`, `mtx_mpyf_fast`) that apply some restrictions.

Precision

4 variants available:

Type	Description
24x24	24-bit inputs, 24-bit output. Available for HiFi3/HiFi3z cores only
32x32	32-bit inputs, 32-bit output
16x16	16-bit inputs, 16-bit output
f	floating point. Requires VFPU core option

Algorithm

For fixed-point routines:

$$z_{m,p} = 2^{lsh} \sum_{n=0}^{N-1} x_{m,n} \cdot y_{n,p}, m = \overline{0...M-1}, p = \overline{0...P-1}$$

For floating point routines:

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

Prototype

```
void mtx_mpy24x24 (  void* pScr,
                     f24*   z,
               const f24*   x,
               const f24*   y,
               int M, int N, int P, int lsh );
void mtx_mpy32x32 (  void* pScr,
                     int32_t*  z,
               const int32_t*  x,
               const int32_t*  y,
               int M, int N, int P, int lsh );
void mtx_mpy16x16 (  void* pScr,
                     int16_t*  z,
               const int16_t*  x,
               const int16_t*  y,
               int M, int N, int P, int lsh );
void mtx_mpyf (        float32_t*   z,
               const float32_t*   x,
               const float32_t*   y,
               int M, int N, int P);
```

```
void mtx_mpy24x24_fast (  f24*  z,
                    const f24* x,
                    const f24* y,
                    int M, int N, int P, int lsh );
void mtx_mpy32x32_fast (int32_t* z,
                  const int32_t* x,
                  const int32_t* y,
                  int M, int N, int P, int lsh );
void mtx_mpy16x16_fast (  int16_t* z,
                    const int16_t* x,
                    const int16_t* y,
                    int M, int N, int P, int lsh );
void mtx_mpyf_fast ( float32_t*   z,
               const float32_t*   x,
               const float32_t*   y,
               int M, int N, int P);
```

Arguments

Type	Name	Size	Description
Input			
`f24, int16_t, int32_t, float32_t`	`x`	`M*N`	input matrix,Q31,Q15, floating point
`f24, int16_t, int32_t, float32_t`	`y`	`N*P`	input matrix `y`. Q31,Q15, floating point.
`int`	`M`		number of rows in matrix `x` and `z`
`int`	`N`		number of columns in matrix x and number of rows in matrix `y`
`int`	`P`		number of columns in matrices `y` and `z`
`int`	`lsh`		left shift applied to the result (applied to the fixed-point functions only)
Output			
`f24, int16_t, int32_t, float32_t`	`z`	`M*P`	output matrix `z`,Q31,Q15, floating point
Temporary			
`void*`	`pScr`		Scratch memory area with size in bytes defined by macros `SCRATCH_MTX_MPY24X24`, `SCRATCH_MTX_MPY32X32`, `SCRATCH_MTX_MPY16X16`

Returned value

none

Restrictions

For regular routines (`mtx_mpy24x24, mtx_mpy32x32, mtx_mpy16x16, mtx_mpyf`):
`x,y,z` should not overlap

For faster routines (`mtx_mpy24x24_fast, mtx_mpy32x32_fast, mtx_mpy16x16_fast, mtx_mpyf_fast`):
`x,y,z` should not overlap
`x,y,z` – aligned on 8-byte boundary
`M,N,P` - multiplies of 4
lsh should be in range:
-31...31 for `mtx_mpy32x32, mtx_mpy32x32_fast, mtx_mpy24x24, mtx_mpy24x24_fast;`
-15...15 for `mtx_mpy16x16, mtx_mpy16x16_fast`

2.6.2 Matrix by Vector Multiply

Description

These functions compute the expression `z = 2^lsh * x * y` for the matrices `x` and vector `y`.
NOTE: `lsh` factor is not relevant for floating point routines.
Two versions of functions available: regular version (`mtx_vecmpy24x24, mtx_vecmpy32x32, mtx_vecmpy16x16, mtx_vecmpyf`) with arbitrary arguments and faster version (`mtx_vecmpy24x24_fast, mtx_vecmpy32x32_fast, mtx_vecmpy16x16_fast, mtx_vecmpyf_fast`) that apply some restrictions.

Precision

4 variants available:

Type	Description
24x24	24-bit inputs, 24-bit output. Available for HiFi3/HiFi3z cores only
32x32	32-bit inputs, 32-bit output
16x16	16-bit inputs, 16-bit output
f	floating point. Requires VFPU core option

Algorithm

For fixed-point routines:

$$z_n = 2^{lsh} \sum_{m=0}^{M-1} x_{n,m} \cdot y_m, n = 0...\overline{N-1}$$

For floating-point routines:

$$z_n = \sum_{m=0}^{M-1} x_{n,m} \cdot y_m, n = 0...\overline{N-1}$$

Prototype

```
void mtx_vecmpy24x24 (  f24*  z,
                  const f24*  x,
                  const f24*  y,
                  int M, int N, int lsh);
void mtx_vecmpy32x32 (  int32_t*  z,
                  const int32_t*  x,
                  const int32_t*  y,
                  int M, int N, int lsh);
void mtx_vecmpy16x16 (  int16_t*  z,
                  const int16_t*  x,
                  const int16_t*  y,
                  int M, int N, int lsh);
void mtx_vecmpyf (   float32_t*  z,
                  const float32_t*  x,
                  const float32_t*  y,
                  int M, int N);

void mtx_vecmpy24x24_fast (  f24*  z,
                  const f24*  x,
                  const f24*  y,
                  int M, int N, int lsh);
void mtx_vecmpy32x32_fast (  int32_t*  z,
                  const int32_t*  x,
                  const int32_t*  y,
                  int M, int N, int lsh);
void mtx_vecmpy16x16_fast (  int16_t*  z,
                  const int16_t*  x,
                  const int16_t*  y,
                  int M, int N, int lsh);
void mtx_vecmpyf_fast (   float32_t*  z,
                  const float32_t*  x,
                  const float32_t*  y,
                  int M, int N);
```

Arguments

Type	Name	Size	Description
Input			
`f24, int32_t, int16_t, float32_t`	`x`	`M*N`	input matrix,Q31, Q15 or floating point

`f24, int32_t, int16_t, float32_t`	`y`	`N`	input vector,Q31, Q15 or floating point
`int`	`M`		number of rows in matrix `x`
`int`	`N`		number of columns in matrix `x`
`int`	`lsh`		left shift applied to the result (applied to the fixed-point functions only)
Output			
`f24, int32_t, int16_t, float32_t`	`z`	`M`	output vector,Q31, Q15 or floating point

Returned value

None

Restrictions

For regular routines (`mtx_vecmpy24x24, mtx_vecmpy32x32, mtx_vecmpy16x16, mtx_vecmpyf`)
`x,y,z` should not overlap

For faster routines (`mtx_vecmpy24x24_fast, mtx_vecmpy32x32_fast, mtx_vecmpy16x16_fast, mtx_vecmpyf_fast`)
`x,y,z` should not overlap
`x,y` aligned on 8-byte boundary
`N` and `M` are multiples of 4
lsh should be in range:
-31...31 for `mtx_vecmpy32x32, mtx_vecmpy32x32_fast, mtx_vecmpy24x24, mtx_vecmpy24x24_fast;`
-15...15 for `mtx_vecmpy16x16, mtx_vecmpy16x16_fast`

2.7 Matrix Decomposition/Inversion

2.7.1 Matrix Inverse

Description

These functions implement in-place matrix inversion by Gauss elimination with full pivoting.
NOTE: user may detect "invalid" or "divide-by-zero" exception in the CPU flags which MAY indicate that inversion results are not accurate. Also it's responsibility of the user to provide valid input matrix for inversion.

Precision

1 variant available:

Type	Description
f	floating point. Requires VFPU core option

Algorithm

$y = x^{-1}$

Prototype

```
void mtx_inv2x2f(float32_t *x);
void mtx_inv3x3f(float32_t *x);
void mtx_inv4x4f(float32_t *x);
```

Matrix dimension, N	Function
2	inv2x2f
3	inv3x3f
4	inv4x4f

Arguments

Type	Name	Size	Description
Input			
float32_t	x	N*N	input matrix
Output			
float32_t	x	N*N	output inverted matrix

Returned value

none

Restrictions

none

2.8 Fitting/Interpolation

2.8.1 Polynomial Approximation

Description

Functions calculate polynomial approximation for all values from given vector. Fixed point functions take polynomial coefficients in Q31 precision.
NOTE:
approximation is calculated like Taylor series that is why overflow may potentially occur if cumulative sum of coefficients given from the last to the first coefficient is bigger that 1. To avoid this negative effect for fixed point routines, all the coefficients may be scaled down and result will be shifted left after all intermediate computations. Amount of this left shift is controlled by `lsh` argument.

Precision

3 variants available:

Type	Description
24x24	24-bit inputs, 24-bit coefficients, 24-bit output. Available for HiFi3/HiFi3z cores only
32x32	32-bit inputs, 32-bit coefficients, 32-bit output.
f	floating point. Requires VFPU core option

Algorithm

$$z_n = \sum_{m=0}^{M} c_m x_n^m, n = \overline{0...N-1}$$

Prototype

```
void vec_poly4_24x24
             (f24 *  z, const f24 *  x,
              const f24 *  c, int lsh, int N );
void vec_poly8_24x24
             (f24 *  z, const f24 *  x,
              const f24 *  c, int lsh, int N );
void vec_poly8f
             (float32_t *  z, const float32_t *  x,
              const float32_t *  c, int N );
void vec_poly4_32x32
             (int32_t *  z, const int32_t *  x,
              const int32_t *  c, int lsh, int N );
void vec_poly8_32x32
              (int32_t *  z, const int32_t *  x,
               const int32_t *  c, int lsh, int N );
void vec_poly4f
             (float32_t *  z, const float32_t *  x,
              const float32_t *  c, int N );
```

Arguments

Type	Name	Size	Description
Input			
`f24, int32_t, float32_t`	`x`	`N`	input data, Q31 or floating point
`f24,int32_t, float32_t`	`c`	`5 or 9`	coefficients (5 coefficients for `vec_poly4_xxx` and 9 coefficients for `vec_poly8_xxx`), Q31 or floating point
`int`	`lsh`		additional left shift for result (for fixed point routines only)
`int`	`N`		length of vectors
Output			
`f24, int32_t, float32_t`	`z`	`N`	result, Q31 or floating point

Returned value

None

Restrictions

`x,c,z` should not overlap

Conditions for optimum performance

`x,c,z` - aligned on 8-byte boundary
`N` - multiple of 2

2.9 Fast Fourier Transforms

FFT functions make floating point, 32x32, 32x16, 24x24, 16x16-bit scaling fast Fourier transforms for complex/real data. Also, they use bit-reversal permutations so spectral data appear in the usual order. They normally use in-place transformations so **input data may be damaged**.

Different types if data scaling are provided by FFT functions. For all types of scaling, the internal representation of the data is the same as the input/output data, except for `*24x24_ie_24p`, `*32x16_ie_24p` functions (see 2.9.6, 2.9.8). For these functions, the internal representation of the data is `complex_fract32`.

Basic scaling modes:

- **dynamic scaling (**`scalingOption = 2`**)**, provides the best accuracy, but has less performance comparing with static scaling;
- **static scaling (**`scalingOption = 3`**),** has more performance but worse accuracy than dynamic scaling.

With dynamic scaling (`scalingOption = 2`), the input data are normalized in the first phase of the FFT, but so that there is no overflow. In subsequent phases, the data are automatically shifted to the right, so that there is no overflow. The function returns a total shift count, which can be negative under certain conditions (i.e. weak input signals).

With static scaling (`scalingOption = 3`), the data are shifted to the right before each FFT phase, the amount of shift is independent of the input data and is chosen so that there is no overflow for any input data.

FFTs 24x24 have additional scaling modes:

- **No scaling (**`scalingOption = 0`**)**, provides the highest performance, but the worst accuracy. To avoid overflow, the input data must be prescaled by the user, so that the maximum and minimum values of the samples in the input array have at least `2 + log2 (N)` spare (signed) bits
- **24-bit scaling (**`scalingOption = 1`**) -** phase there is no normalization of the input signal, this gives a small increase in performance in comparison with `scalingOption = 2`. This mode is recommended for normalized input data. If input signal is small than quality will degrade.

Example of prescaling data for `scalingOpt = 0`:

```
// const f24 *x  - pointer to input complex data
f24 tmp[2*N];    // temporary buffer
int s = 30 - XT_NSA(N) ;              //  1+log2(N);
int bexp = vec_bexp24(x, 2*N);    //  calculate block exponent
s = (bexp < s)? 0: s - bexp;              // calculate shift
vec_shift24x24(tmp, x, -s, 2*N);   // right shift data
fft_cplx24x24(     y,      tmp,  h, 0); // call fft function
```

FFT/IFFT functions family with improved memory efficiency (`fft_cplx<prec>_ie`, `fft_real<prec>_ie`, `fft_cplx<prec>_ie_24p`, `fft_real<prec>_ie_24p`) as well as floating point FFT functions[2] expose smaller program- and constant data memory footprint. They differ from regular FFT/IFFT functions in the following aspects:

- cycles performance is compromised in favor of memory efficiency
- 24x24 and 32x16 use static scaling method , 32x32 and 16x16 FFTs uses allows dynamic scaling as well
- twiddle factor tables are provided by user. A single table may be shared between FFTs/IFFTs of varying size (see para 4.2)
- 24-bit packed format is used for input/output/temporary data storage were applicable

All fixed-point FFT functions (including scaling and non-scaling) return total number of right shifts (`t`) occurred during all stages. Floating point FFTs do not make additional scaling so they always return 0 to indicate this fact. So, FFT/IFFT output will be scaled by 2^t. Library functions from 2.5.4help to convert results to desired scale or Q-representation. In these computations you have to take into account the fact that FFT→IFFT chain amplifies signal by the length of FFT `N` for complex transforms and by `N/2` for real transforms.
For example, consider processing chain:
`y=FFT(x) → w=some_processing(y) → z=IFFT(w)` where `N` is the length of FFT, FFT returns total shift amount t_{FFT} and IFFT returns t_{IFFT}.
To move `z` to the same scale as `x` you have to shift it by:
$t_{FFT} + t_{IFFT} - \log_2(N) \equiv t_{FFT} + t_{IFFT} - (30 - \mathrm{scl\_bexp32}(N))$

Alternatively, you may treat it as changing Q-representation. For example, DCT functions (with length 32) always return total number of shifts equals to $\log_2(32)=5$. So, if its input is Q31, output will be in Q26.

The table below summarizes how number of right shifts depends on selected scaled option.

Scaling option	FFT functions family	Returned number of right shifts
0	2D DCT	0
2	all FFT functions	depends on input data
3	FFT/IFFT on complex data	`log2(N)+1`
3	FFT/IFFT on real data, DCT	`log2(N)+1`

There are limited combinations of precision, scaling options and restrictions on the dynamic range of the input signal available:

Precision	Scaling options	Restrictions on the dynamic range of the input signal
FFT/IFFT		
`cplx24x24`, `real24x24`	0 – no scaling 1 – 24-bit scaling 2 – 32-bit scaling on the first stage and 24-bit scaling later 3 – fixed scaling before each stage	Input signal < 2^23/(2*N), N - FFT size None None None
`cplx32x16`	3 – fixed scaling before each stage	None
`cplx32x32`	2 – 32-bit dynamic scaling 3 – fixed scaling before each stage	None
`cplx16x16`	2 – 16-bit dynamic scaling 3 – fixed scaling before each stage	None

[2] Floating point FFT available only with improved memory efficiency API

Precision	Scaling options	Restrictions on the dynamic range of the input signal
`cplx16x16_ie`	2 – 16-bit dynamic scaling	None
`cplx24x24_ie`	3 – fixed scaling before each stage	None
`cplx32x16_ie`	3 – fixed scaling before each stage	None
`cplx32x32_ie`	2 – 32-bit dynamic scaling 3 – fixed scaling before each stage	None
`real32x16`	3 – fixed scaling before each stage	None
`real32x32`	2 – 32-bit dynamic scaling 3 – fixed scaling before each stage	None
`real16x16`	2 – 16-bit dynamic scaling 3 – fixed scaling before each stage	None
`real16x16_ie`	2 – 16-bit dynamic scaling	None
`real32x16_ie`	3 – fixed scaling before each stage	None
`real32x32_ie`	2 – 32-bit dynamic scaling 3 – fixed scaling before each stage	None
`real24x24_ie`	3 – fixed scaling before each stage	None
`real32x16_ie_24p`	3 – fixed scaling before each stage	None
`rea24x24_ie_24p`	1 – 24-bit scaling	None
DCT		

`dct_24x24,`
`dct_32x16,`
`dct_32x32,`
`dct_16x16,`
`dct4_24x24,`
`dct4_32x16,`
`dct4_32x32,`
`mdct_24x24,`
`mdct_32x16,`
`mdct_32x32,`
`imdct_24x24,`
`imdct_32x16,`
`imdct_32x32`
	3 – fixed scaling before each stage	None

`dct2d_8x16`	0 – no scaling	None
`idct2d_16x8`	0 – no scaling	None

2.9.1 FFT on Complex Data

Description

These functions make FFT on complex data.
NOTES:
1. Bit-reversing permutation is done here.
2. FFT runs in-place algorithm so INPUT DATA WILL APPEAR DAMAGED after the call
3. 32x32 FFT supports mixed radix transforms

Precision

4 variants available:

Type	Description
24x24	24-bit input/outputs, 24-bit twiddles. Available for HiFi3/HiFi3z cores only
32x16	32-bit input/outputs, 16-bit twiddles
32x32	32-bit input/outputs, 32-bit twiddles
16x16	16-bit input/outputs, 16-bit twiddles

Algorithm

$$y = FFT(x)$$

Prototype

```
int fft_cplx24x24(
              f24* y, f24* x,
              fft_handle_t h, int scalingOption)
int fft_cplx32x16(
              int32_t* y, int32_t* x,
              fft_handle_t h, int scalingOption)
```

```
int fft_cplx32x32(
              int32_t* y, int32_t* x,
              fft_handle_t h, int scalingOption)
int fft_cplx16x16(
              int16_t* y, int16_t* x,
              fft_handle_t h, int scalingOption)
```

FFT handles:

N	24x24	32x16	32x16	32x32
16	cfft24_16	cfft16_16	cfft16_16	cfft32_16
32	cfft24_32	cfft16_32	cfft16_32	cfft32_32
64	cfft24_64	cfft16_64	cfft16_64	cfft32_64
128	cfft24_128	cfft16_128	cfft16_128	cfft32_128
256	cfft24_256	cfft16_256	cfft16_256	cfft32_256
512	cfft24_512	cfft16_512	cfft16_512	cfft32_512
1024	cfft24_1024	cfft16_1024	cfft16_1024	cfft32_1024
2048	cfft24_2048	cfft16_2048	cfft16_2048	cfft32_2048
4096	cfft24_4096	cfft16_4096	cfft16_4096	cfft32_4096

FFT handles for mixed radix transforms (for 32x32 only):

N	32x32	N	32x32	N	32x32
12	cnfft32_12	144	cnfft32_144	384	cnfft32_384
24	cnfft32_24	160	cnfft32_160	400	cnfft32_400
36	cnfft32_36	180	cnfft32_180	432	cnfft32_432
48	cnfft32_48	192	cnfft32_192	480	cnfft32_480
60	cnfft32_60	200	cnfft32_200	540	cnfft32_540
72	cnfft32_72	216	cnfft32_216	576	cnfft32_576
80	cnfft32_80	240	cnfft32_240	600	cnfft32_600
96	cnfft32_96	288	cnfft32_288	768	cnfft32_768
100	cnfft32_100	300	cnfft32_300	960	cnfft32_960
108	cnfft32_108	324	cnfft32_324		
120	cnfft32_120	360	cnfft32_360		

, where N – FFT size

Arguments

Type	Name	Size	Description
Input			
f24, int32_t or int16_t	x	2*N	complex input signal. Real and imaginary data are interleaved and real data goes first
fft_handle_t	h		handle to specific FFT tables
int	scalingOption		scaling option (see table in para 2.9)
Output			
f24, int32_t or int16_t	y	2*N	output spectrum. Real and imaginary data are interleaved and real data goes first

Returned value total number of right shifts occurred during scaling procedure

Restrictions

x,y should not overlap

x,y aligned on a 8-bytes boundary

2.9.2 FFT on Real Data

Description

These functions make FFT on real data forming half of spectrum

NOTES:

1. Bit-reversing reordering is done here.
2. FFT runs in-place algorithm so INPUT DATA WILL APPEAR DAMAGED after the call.

3. Real data FFT function calls `fft_cplx()` to apply complex FFT of size `N/2` to input data and then transforms the resulting spectrum.
4. 32x32 FFT supports mixed radix transforms

Precision

4 variants available:

Type	Description
24x24	24-bit input/outputs, 24-bit twiddles. Available for HiFi3/HiFi3z cores only
32x16	32-bit input/outputs, 16-bit twiddles
32x32	32-bit input/outputs, 32-bit twiddles
16x16	16-bit input/outputs, 16-bit twiddles

Algorithm

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

Prototype

```
int fft_real24x24(
              f24* y, f24* x,
              fft_handle_t h, int scalingOpt)
int fft_real32x16(
              int32_t* y, int32_t* x,
              fft_handle_t h, int scalingOpt)
int fft_real32x32(
              int32_t* y, int32_t* x,
              fft_handle_t h, int scalingOpt)
int fft_real16x16(
              int16_t* y, int16_t* x,
              fft_handle_t h, int scalingOpt)
```

FFT handles:

N	24x24	32x16	32x16	32x32
32	rfft24_32	rfft16_32	rfft16_32	rfft32_32
64	rfft24_64	rfft16_64	rfft16_64	rfft32_64
128	rfft24_128	rfft16_128	rfft16_128	rfft32_128
256	rfft24_256	rfft16_256	rfft16_256	rfft32_256
512	rfft24_512	rfft16_512	rfft16_512	rfft32_512
1024	rfft24_1024	rfft16_1024	rfft16_1024	rfft32_1024
2048	rfft24_2048	rfft16_2048	rfft16_2048	rfft32_2048
4096	rfft24_4096	rfft16_4096	rfft16_4096	rfft32_4096
8192	rfft24_8192	rfft16_8192	rfft16_8192	rfft32_8192

FFT handles for mixed radix transforms (for 32x32 only):

N	32x32	N	32x32	N	32x32
12	rnfft32_12	144	rnfft32_144	480	rnfft32_480
24	rnfft32_24	180	rnfft32_180	540	rnfft32_540
30	rnfft32_30	192	rnfft32_192	576	rnfft32_576
36	rnfft32_36	216	rnfft32_216	720	rnfft32_720
48	rnfft32_48	240	rnfft32_240	768	rnfft32_768
60	rnfft32_60	288	rnfft32_288	960	rnfft32_960
72	rnfft32_72	300	rnfft32_300	1152	rnfft32_1152
90	rnfft32_90	324	rnfft32_324	1440	rnfft32_1440
96	rnfft32_96	360	rnfft32_360	1536	rnfft32_1536
108	rnfft32_108	384	rnfft32_384	1920	rnfft32_1920
120	rnfft32_120	432	rnfft32_432		

`, where N –` FFT size

Arguments

Type	Name	Size	Description
Input			

`f24, int32_t or int16_t`	`x`	`N`	input signal
`fft_handle_t`	`h`		handle to specific FFT tables
`int`	`scalingOpt`		scaling option (see table in para 2.9)
Output			
`f24 or int16_t`	`y`	`(N/2+1)*2`	output spectrum (positive side). Real and imaginary data are interleaved and real data goes first

Returned value

total number of right shifts occurred during scaling procedure

Restrictions

Arrays should not overlap
`x,y` - aligned on a 8-bytes boundary

2.9.3 Inverse FFT on Complex Data

Description

These functions make inverse FFT on complex data.
NOTES:
1. Bit-reversing reordering is done here.
2. FFT runs in-place algorithm so **INPUT DATA WILL APPEAR DAMAGED** after call
3. 32x32 FFT supports mixed radix transforms

Precision

4 variants available:

Type	Description
24x24	24-bit input/outputs, 24-bit twiddles. Available for HiFi3/HiFi3z cores only
32x16	32-bit input/outputs, 16-bit twiddles
32x32	32-bit input/outputs, 32-bit twiddles
16x16	16-bit input/outputs, 16-bit twiddles

Algorithm

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

Prototype

```
int ifft_cplx24x24(
                f24* y, f24* x,
                fft_handle_t h, int scalingOption)
int ifft_cplx32x16(
                int32_t * y, int32_t* x,
                fft_handle_t h, int scalingOption)
int ifft_cplx32x32(
                int32_t * y, int32_t* x,
                fft_handle_t h, int scalingOption)
int ifft_cplx16x16(
                int16_t* y, int16_t* x, fft_handle_t h, int scalingOption)
```

FFT handles:

N	24x24	32x16	32x16	32x32
16	`cifft24_16`	`cifft16_16`	`cifft16_16`	`cifft32_16`
32	`cifft24_32`	`cifft16_32`	`cifft16_32`	`cifft32_32`
64	`cifft24_64`	`cifft16_64`	`cifft16_64`	`cifft32_64`
128	`cifft24_128`	`cifft16_128`	`cifft16_128`	`cifft32_128`
256	`cifft24_256`	`cifft16_256`	`cifft16_256`	`cifft32_256`
512	`cifft24_512`	`cifft16_512`	`cifft16_512`	`cifft32_512`
1024	`cifft24_1024`	`cifft16_1024`	`cifft16_1024`	`cifft32_1024`
2048	`cifft24_2048`	`cifft16_2048`	`cifft16_2048`	`cifft32_2048`
4096	`cifft24_4096`	`cifft16_4096`	`cifft16_4096`	`cifft32_4096`

FFT handles for mixed radix transforms (for 32x32 only):

N	32x32	N	32x32	N	32x32
12	`cinfft32_12`	144	`cinfft32_144`	384	`cinfft32_384`
24	`cinfft32_24`	160	`cinfft32_160`	400	`cinfft32_400`
36	`cinfft32_36`	180	`cinfft32_180`	432	`cinfft32_432`

48	`cinfft32_48`	192	`cinfft32_192`	480	`cinfft32_480`
60	`cinfft32_60`	200	`cinfft32_200`	540	`cinfft32_540`
72	`cinfft32_72`	216	`cinfft32_216`	576	`cinfft32_576`
80	`cinfft32_80`	240	`cinfft32_240`	600	`cinfft32_600`
96	`cinfft32_96`	288	`cinfft32_288`	768	`cinfft32_768`
100	`cinfft32_100`	300	`cinfft32_300`	960	`cinfft32_960`
108	`cinfft32_108`	324	`cinfft32_324`		
120	`cinfft32_120`	360	`cinfft32_360`		

`, where N -` IFFT size

Arguments

Type	Name	Size	Description
Input			
`f24, int32_t or int16_t`	`x`	`2*N`	input spectrum. Real and imaginary data are interleaved and real data goes first
`fft_handle_t`	`h`		handle to specific FFT tables
`int`	`scalingOpt`		scaling option (see table in para 2.9)
Output			
`f24, int32_t or int16_t`	`y`	`2*N`	complex output signal. Real and imaginary data are interleaved and real data goes first

Returned value total number of right shifts occurred during scaling procedure

Restrictions

`x,y` - should not overlap
`x,y` - aligned on 8-bytes boundary

2.9.4 Inverse FFT Forming Real Data

Description

These functions make inverse FFT on half spectral data forming real data samples
NOTES:
1. Bit-reversing reordering is done here.
2. IFFT runs in-place algorithm so INPUT DATA WILL APPEAR DAMAGED after call.
3. Inverse FFT function for real signal transforms the input spectrum and then calls `ifft_cplx()` with FFT size set to `N/2`.
4. 32x32 FFT supports mixed radix transforms

Precision

4 variants available:

Type	Description
24x24	24-bit input/outputs, 24-bit twiddles. Available for HiFi3/HiFi3z cores only
32x16	32-bit input/outputs, 16-bit twiddles
32x32	32-bit input/outputs, 32-bit twiddles
16x16	16-bit input/outputs, 16-bit twiddles

Algorithm

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

Prototype

```
int ifft_real24x24(
              f24* y, f24* x,
              fft_handle_t h, int scalingOpt)
int ifft_real32x16(
              int32_t * y, int32_t* x,
              fft_handle_t h, int scalingOpt)
int ifft_real32x32(
              int32_t * y, int32_t* x,
              fft_handle_t h, int scalingOpt)
int ifft_real16x16(
              int16_t* y, int16_t* x,
              fft_handle_t h, int scalingOpt)
```

FFT handles:

N	24x24	32x16	32x16	32x32
32	rifft24_32	rifft16_32	rifft16_32	rifft32_32
64	rifft24_64	rifft16_64	rifft16_64	rifft32_64
128	rifft24_128	rifft16_128	rifft16_128	rifft32_128
256	rifft24_256	rifft16_256	rifft16_256	rifft32_256
512	rifft24_512	rifft16_512	rifft16_512	rifft32_512
1024	rifft24_1024	rifft16_1024	rifft16_1024	rifft32_1024
2048	rifft24_2048	rifft16_2048	rifft16_2048	rifft32_2048
4096	rifft24_4096	rifft16_4096	rifft16_4096	rifft32_4096
8192	rifft24_8192	rifft16_8192	rifft16_8192	rifft32_8192

FFT handles for mixed radix transforms (for 32x32 only) :

N	32x32	N	32x32	N	32x32
12	rinfft32_12	144	rinfft32_144	480	rinfft32_480
24	rinfft32_24	180	rinfft32_180	540	rinfft32_540
30	rinfft32_30	192	rinfft32_192	576	rinfft32_576
36	rinfft32_36	216	rinfft32_216	720	rinfft32_720
48	rinfft32_48	240	rinfft32_240	768	rinfft32_768
60	rinfft32_60	288	rinfft32_288	960	rinfft32_960
72	rinfft32_72	300	rinfft32_300	1152	rinfft32_1152
90	rinfft32_90	324	rinfft32_324	1440	rinfft32_1440
96	rinfft32_96	360	rinfft32_360	1536	rinfft32_1536
108	rinfft32_108	384	rinfft32_384	1920	rinfft32_1920
120	rinfft32_120	432	rinfft32_432		

,where N - IFFT size

Arguments

Type	Name	Size	Description
Input			
f24, int32_t or int16_t	x	(N/2+1)*2	input spectrum. Real and imaginary data are interleaved and real data goes first
fft_handle_t	h		handle to specific FFT tables
int	scalingOpt		scaling option (see table in para 2.9)
Output			
f24, int32_t or int16_t	y	N	real output signal

Returned value

total number of right shifts occurred during scaling procedure

Restrictions

x,y should not overlap
x,y - aligned on 8-bytes boundary

2.9.5 FFT on Complex Data with Optimized Memory Usage

Description

These functions make FFT on complex data with optimized memory usage
NOTES:
1. Bit-reversing permutation is done here.
2. FFT runs in-place algorithm so INPUT DATA WILL APPEAR DAMAGED after the call
3. FFT of size N may be supplied with constant data (twiddle factors) of a larger-sized FFT = N*twdstep.

Precision

5 variants available:

Type	Description
24x24	24-bit input/outputs, 24-bit twiddles. Available for HiFi3/HiFi3z cores only
32x16	32-bit input/outputs, 16-bit twiddles
32x32	32-bit input/outputs, 32-bit twiddles

`16x16`	16-bit input/outputs, 16-bit twiddles
`f`	floating point. Requires VFPU core option

Algorithm $y = FFT(x)$

Prototype

```
int fft_cplx24x24_ie(
              complex_fract32* y, complex_fract32* x,
              const complex_fract32* twd,
              int twdstep, int N, int scalingOpt);
int fft_cplx32x16_ie(
              complex_fract32* y, complex_fract32* x,
              const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int fft_cplx32x32_ie(
              complex_fract32* y, complex_fract32* x,
              const complex_fract32* twd,
              int twdstep, int N, int scalingOpt);
int fft_cplx16x16_ie(
              complex_fract16* y, complex_fract16* x,
              const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int fft_cplxf_ie (
             complex_float * y, complex_float * x,
             const complex_float* twd,
             int twdstep, int N );
```

Arguments

Type	Name	Size	Description
Input			
`complex_fract16,` `complex_fract32,` `complex_float`	`x`	`N`	complex input signal. Real and imaginary data are interleaved and real data goes first
`complex_fract32,` `complex_fract16,` `complex_float`	`twd`	`N*3/4*twdstep`	twiddle factor table of a complex-valued FFT of size `N*twdstep`
`int`	`twdstep`		twiddle step
`int`	`N`		FFT size
`int`	`scalingOpt`		scaling option (see table in para 2.9) , not applicable to the floating point function
Output			
`complex_fract16,` `complex_fract32,` `complex_float`	`y`	`N`	output spectrum. Real and imaginary data are interleaved and real data goes first

Returned value total number of right shifts occurred during scaling procedure. Floating function always return 0

Restrictions

`x,y` should not overlap

`x,y` - aligned on a 8-bytes boundary

2.9.6 FFT on Real Data with Optimized Memory Usage

Description

These functions make FFT on real data forming half of spectrum with optimized memory usage

NOTES:

1. Bit-reversing reordering is done here.
2. FFT runs in-place algorithm so INPUT DATA WILL APPEAR DAMAGED after the call.
3. FFT functions use input and output buffers for temporary storage of intermediate 32-bit data, so FFT functions with 24-bit packed I/O (Nx3-byte data) require that the buffers are large enough to keep Nx4-byte data.
4. FFT of size `N` may be supplied with constant data (twiddle factors) of a larger-sized FFT = N*twdstep

Precision

7 variants available:

Type	Description

`24x24`	24-bit input/outputs, 24-bit twiddles. Available for HiFi3/HiFi3z cores only
`32x16`	32-bit input/outputs, 16-bit twiddles
`32x32`	32-bit input/outputs, 32-bit twiddles
`16x16`	16-bit input/outputs, 16-bit twiddles
`24x24_ie_24p`	24-bit packed input/outputs, 24-bit data, 24-bit twiddles. Available for HiFi3/HiFi3z cores only
`32x16_ie_24p`	24-bit packed input/outputs, 32-bit data, 16-bit twiddles. Available for HiFi3/HiFi3z cores only
`f`	floating point. Requires VFPU core option

Algorithm

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

Prototype

```
int fft_real24x24_ie(
              complex_fract32* y, f24* x, const complex_fract32* twd,
              int twdstep, int N, int scalingOpt);
int fft_real32x16_ie(
              complex_fract32* y, int32_t* x, const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int fft_real32x32_ie(
              complex_fract32* y, int32_t* x, const complex_fract32* twd,
              int twdstep, int N, int scalingOpt);
int fft_real16x16_ie(
              complex_fract16* y, int16_t* x, const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int fft_realf_ie(
              complex_float* y, float32_t* x, const complex_float* twd,
              int twdstep, int N);
int fft_real24x24_ie_24p(
              uint8_t* y, uint8_t* x, const complex_fract32* twd,
              int twdstep, int N, int scalingOpt);
int fft_real32x16_ie_24p(uint8_t* y, uint8_t* x, const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
```

Arguments

Type	Name	Size	Allocated Size	Description
Input				
`int16_t, f24, int32_t, float32_t`	`x`	`N`	`N`	input signal
`uint8_t`		`3*N`	`4*N+8`	
`complex_fract32, complex_fract16, complex_float`	`twd`	`N*3/4 *twdstep`		twiddle factor table of a complex-valued FFT of size `N*twdstep`
`int`	`twdstep`			twiddle step
`int`	`N`			FFT size
`int`	`scalingOpt`			scaling option (see table in para 2.9) , not applicable to the floating point function
Output				
`complex_fract16, complex_fract32, complex_float`	`y`	`N/2+1`	`N/2+1`	output spectrum (positive side). Real and imaginary data are interleaved and real data goes first
`uint8_t`		`3*(N+2)`	`4*N+8`	

Returned value

total number of right shifts occurred during scaling procedure. Floating function always return 0

Restrictions

Arrays should not overlap
`x,y` - aligned on a 8-bytes boundary
N must be in powers of 2

2.9.7 Inverse FFT on Complex Data with Optimized Memory Usage

Description

These functions make inverse FFT on complex data with optimized memory usage
NOTES:
1. Bit-reversing permutation is done here.
2. FFT runs in-place algorithm so INPUT DATA WILL APPEAR DAMAGED after the call
3. FFT of size `N` may be supplied with constant data (twiddle factors) of a larger-sized FFT = `N*twdstep`.

Precision

5 variants available:

Type	Description
`24x24`	24-bit input/outputs, 24-bit twiddles. Available for HiFi3/HiFi3z cores only
`32x16`	32-bit input/outputs, 16-bit twiddles
`32x32`	32-bit input/outputs, 32-bit twiddles
`16x16`	16-bit input/outputs, 16-bit twiddles
`f`	floating point. Requires VFPU core option

Algorithm

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

Prototype

```
int ifft_cplx24x24_ie(
              complex_fract32* y, complex_fract32* x,
              const complex_fract32* twd,
              int twdstep, int N, int scalingOpt);
int ifft_cplx32x16_ie(
              complex_fract32* y, complex_fract32* x,
              const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int ifft_cplx32x32_ie(
              complex_fract32* y, complex_fract32* x,
              const complex_fract32* twd,
              int twdstep, int N, int scalingOpt);
int ifft_cplx16x16_ie(
              complex_fract16* y, complex_fract16* x,
              const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int ifft_cplxf_ie(
              complex_float* y, complex_float * x,
              const complex_float * twd,
              int twdstep, int N);
```

Arguments

Type	Name	Size	Description
Input			

`complex_fract16,`
`complex_float`
`complex_fract32,`
`complex_float`
	`x`	`N`	complex input signal. Real and imaginary data are interleaved and real data goes first

`complex_fract32,`
`complex_fract16,`
`complex_float`
	`twd`	`N*3/4*twdstep`	twiddle factor table of a complex-valued FFT of size `N*twdstep`

`int`	`twdstep`		twiddle step
`int`	`N`		FFT size
`int`	`scalingOpt`		scaling option (see table in para 2.9) , not applicable to the floating point function
Output			

`complex_fract16,`
`complex_fract32,`
`complex_float`
	`y`	`N`	output spectrum. Real and imaginary data are interleaved and real data goes first

Returned value

total number of right shifts occurred during scaling procedure. Floating function always return 0

Restrictions

`x,y` should not overlap
`x,y` - aligned on a 8-bytes boundary

2.9.8 Inverse FFT on Real Data with Optimized Memory Usage

Description

These functions make inverse FFT on real data from half of spectrum with optimized memory usage
NOTES:
1. Bit-reversing reordering is done here.
2. FFT runs in-place algorithm so INPUT DATA WILL APPEAR DAMAGED after the call.
3. FFT functions use input and output buffers for temporary storage of intermediate 32-bit data, so FFT functions with 24-bit packed I/O (`Nx3`-byte data) require that the buffers are large enough to keep `Nx4`-byte data.
4. FFT of size `N` may be supplied with constant data (twiddle factors) of a larger-sized FFT = `N*twdstep`

Precision

7 variants available:

Type	Description
`24x24`	24-bit input/outputs, 24-bit twiddles. Available for HiFi3/HiFi3z cores only
`32x16`	32-bit input/outputs, 16-bit twiddles
`32x32`	32-bit input/outputs, 32-bit twiddles
`16x16`	16-bit input/outputs, 16-bit twiddles
`24x24_ie_24p`	24-bit packed input/outputs, 24-bit data, 24-bit twiddles. Available for HiFi3/HiFi3z cores only
`32x16_ie_24p`	24-bit packed input/outputs, 32-bit data, 16-bit twiddles. Available for HiFi3/HiFi3z cores only
`f`	floating point. Requires VFPU core option

Algorithm

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

Prototype

```
int ifft_real24x24_ie(
              f24* y, complex_fract32* x, const complex_fract32* twd,
              int twdstep, int N, int scalingOpt);
int ifft_real32x16_ie(
              int32_t* y, complex_fract32* x, const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int ifft_real32x32_ie(
              int32_t* y, complex_fract32* x, const complex_fract32* twd,
              int twdstep, int N, int scalingOpt);
int ifft_real16x16_ie(
              int32_t* y, complex_fract16* x, const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
int ifft_realf_ie(
              float32_t* y, complex_float * x, const complex_float* twd,
              int twdstep, int N);
int ifft_real24x24_ie_24p(
              uint8_t* y, uint8_t* x, const complex_fract32* twd,
              int twdstep, int N, int scalingOpt);
int ifft_real32x16_ie_24p(
              uint8_t* y, uint8_t* x, const complex_fract16* twd,
              int twdstep, int N, int scalingOpt);
```

Arguments

Type	Name	Size	Allocated Size	Description
Input				
`complex_fract16`, `complex_fract32`, `complex_float`	`x`	`N/2+1`	`N/2+1`	input spectrum (positive side). Real and imaginary data are interleaved and real data goes first
`uint8_t`		`3*(N+2)`	`4*N+8`	
`complex_fract32`, `complex_fract16`, `complex_float`	`twd`	`N*3/4*twdstep`		twiddle factor table of a complex-valued FFT of size `N*twdstep`
`int`	`twdstep`			twiddle step
`int`	`N`			FFT size

`int`	`scalingOpt`			scaling option (see table in para 2.9) , not applicable to the floating point function
Output				
`f24, int16_t, float32_t`	`y`	`N`	`N`	output real signal
`uint8_t`		`3*N`	`4*N+8`	

Returned value total number of right shifts occurred during scaling procedure. Floating function always return 0

Restrictions Arrays should not overlap
`x,y` - aligned on a 8-bytes boundary

2.9.9 Discrete Cosine Transform

Description These functions apply DCT (Type II, Type IV) to input
NOTE:
DCT runs in-place algorithm so **INPUT DATA WILL APPEAR DAMAGED** after the call.

Precision 5 variants available:

Type	Description
24x24	24-bit input/outputs, 24-bit twiddles. Available for HiFi3/HiFi3z cores only
32x16	32-bit input/outputs, 16-bit twiddles
32x32	32-bit input/outputs, 32-bit twiddles
16x16	16-bit input/outputs, 16-bit twiddles
f	floating point. Requires VFPU core option

Algorithm

DCT Type II: $y_{k=0..N-1} = \sum_{n=0}^{N-1} x_n \cdot \cos\left(\frac{\pi}{N} \cdot (n+0.5) \cdot k\right), n = \overline{0...N-1}$

DCT Type IV: $y_{k=0..N-1} = \sum_{n=0}^{N-1} x_n \cdot \cos\left(\frac{\pi}{N} \cdot (n+0.5) \cdot [k+0.5]\right), n = \overline{0...N-1}$

Prototype (DCT Type II)

```
int dct_24x24(f24    * y, f24    * x, dct_handle_t h, int scalingOpt);
int dct_32x16(int32_t* y, int32_t* x, dct_handle_t h, int scalingOpt);
int dct_32x32(int32_t* y, int32_t* x, dct_handle_t h, int scalingOpt);
int dct_16x16(int16_t* y, int16_t* x, dct_handle_t h, int scalingOpt);
int dctf     ( float32_t * y,float32_t * x, dct_handle_t h  );
```

DCT-II handles :

N	32x32, 24x24	N	32x16, 16x16	N	floating point
32	`dct2_32_32`	32	`dct2_16_32`	32	`dct2_f_32`
64	`dct2_32_64`	64	`dct2_16_64`	64	`dct4_f_64`

`,where N -` DCT size

Arguments

Type	Name	Size	Description
Input			
`int16_t, int32_t, f24, float32_t`	`x`	`N`	input signal
`dct_handle_t`	`h`		DCT-II handle
`int`	`scalingOpt`		scaling option (see table in para 2.9), not applicable to the floating point function
Output			

int16_t, int32_t, f24, float32_t	y	N	output of transform

Prototype (DCT Type IV)

```
int dct4_24x24(f24    * y, f24    * x, dct_handle h, int scalingOpt);
int dct4_32x16(int32_t* y, int32_t* x, dct_handle h, int scalingOpt);
int dct4_32x32(int32_t* y, int32_t* x, dct_handle h, int scalingOpt);
```

DCT-IV handles:

N	32x32, 24x24	N	32x16
32	dct4_32_32	32	dct4_16_32
64	dct4_32_64	64	dct4_16_64
128	dct4_32_128	128	dct4_16_128
256	dct4_32_256	256	dct4_16_256
512	dct4_32_512	512	dct4_16_512

, where N - DCT size

Arguments

Type	Name	Size	Description
Input			
int32_t, f24	x	N	input signal
dct_handle_t	h		DCT-IV handle
int	scalingOpt		scaling option (see table in para 2.9), not applicable to the floating point function
Output			
int16_t, int32_t, f24, float32_t	y	N	output of transform

Returned value total number of right shifts occurred during scaling procedure (0 for floating point function)

Restrictions

x,y should not overlap
x,y - aligned on 8-bytes boundary

2.9.10 Modified Discrete Cosine Transform

Description These functions apply Modified DCT to input (convert 2N real data to N spectral components) and make inverse conversion forming 2N numbers from N inputs.

NOTE:
MDCT runs in-place algorithm so **INPUT DATA WILL APPEAR DAMAGED** after the call.

Precision 3 variants available:

Type	Description
24x24	24-bit input/outputs, 24-bit twiddles. Available for HiFi3/HiFi3z cores only
32x16	32-bit input/outputs, 16-bit twiddles
32x32	32-bit input/outputs, 32-bit twiddles

Algorithm

MDCT: $y_{k=0..N-1} = \sum_{n=0}^{N-1} x_n \cdot \cos\left(\frac{\pi}{N} \cdot (n+0.5) \cdot k\right), n = \overline{0...N-1}$

Prototype (direct transform)

```
int mdct_24x24(    f24* x,     f24* y, dct_handle_t h, int scalingOpt);
int mdct_32x16(int32_t* x, int32_t* y, dct_handle_t h, int scalingOpt);
int mdct_32x32(int32_t* x, int32_t* y, dct_handle_t h, int scalingOpt);
```

MDCT/IMDCT handles:

N	32x32, 24x24	N	32x16
32	`mdct_32_32`	32	`mdct_16_32`
64	`mdct_32_64`	64	`mdct_16_64`
128	`mdct_32_128`	128	`mdct_16_128`
256	`mdct_32_256`	256	`mdct_16_256`
512	`mdct_32_512`	512	`mdct_16_512`

`, where N -` MDCT/IMDCT size

Arguments

Type	Name	Size	Description
Input			
`int32_t, f24`	`x`	`N`	input signal
`dct_handle_t`	`h`		MDCT handle
`int`	`scalingOpt`		scaling option (see table in para 2.9)
Output			
`int32_t, f24`	`y`	`2*N`	output of transform

Algorithm

Inverse MDCT: $y_{k=0..N-1} = \sum_{n=0}^{N-1} x_n \cdot \cos\left(\frac{\pi}{N} \cdot (n+0.5)\cdot(k+0.5)\right), n=\overline{0...N-1}$

Prototype (inverse transform)

```
int imdct_24x24(    f24* y,     f24* x, dct_handle h, int scalingOpt);
int imdct_32x16(int32_t* y, int32_t* x, dct_handle h, int scalingOpt);
int imdct_32x32(int32_t* y, int32_t* x, dct_handle h, int scalingOpt);
```

Arguments

Type	Name	Size	Description
Input			
`int32_t, f24`	`x`	`2*N`	input signal
`dct_handle_t`	`h`		IMDCT handle
`int`	`scalingOpt`		scaling option (see table in para 2.9)
Output			
`int32_t, f24`	`y`	`N`	output of transform

Returned value

total number of right shifts occurred during scaling procedure (0 for floating point function)

Restrictions

`x,y` should not overlap
`x,y` - aligned on 8-bytes boundary

2.9.11 2D Discrete Cosine Transform

Description

These functions apply DCT (Type II) to the series of `L` input blocks of `NxN` pixels.

Precision

1 variant available:

Type	Description
8x16	8-bit unsigned input, 16-bit signed output

Algorithm

Algorithm uses ITU-T T.81 (JPEG compression) DCT-II definition with bias 128 and left-to-right, top-to-bottom orientation.

$$y_{n,m}^{(l)} = \frac{1}{4} C_m C_n \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} (x_{j,i}^{(l)} - 128) \cos\frac{(2i+1)m\pi}{2N} \cos\frac{(2j+1)n\pi}{2N}, l=\overline{0...L-1}$$

$$C_k = \begin{cases} 1 & ,k \neq 0 \\ 1/\sqrt{2} & ,k \equiv 0 \end{cases}$$

$x_{j,i}^{(l)} = x[l \cdot N \cdot N + (N \cdot j + i)]$ - pixel from j-th row, i-th column from l-th NxN block

Prototype

```
int dct2d_8x16(int16_t* y, uint8_t * x, dct_handle_t h, int L, int scalingOpt);
```

2D-DCT handles:

N	8x16
8	`dct2d_16_8`

`, where N -` DCT size

Arguments

Type	Name	Size	Description
Input			
`uint8_t`	`x`	`N*N*L`	input pixels: `L NxN` blocks
`dct_handle_t`	`h`		DCT handle
`int`	`L`		number of input blocks
`int`	`scalingOpt`		scaling option (see table in para 2.9), should be 0
Output			
`int16_t`	`y`	`N*N*L`	output of transform: `L NxN` blocks

Returned value 0

Restrictions

`x,y` should not overlap

`x,y` - aligned on 8-bytes boundary

2.9.12 2D Inverse Discrete Cosine Transform

Description These functions apply inverse DCT (Type II) to the series of `L` input blocks of `NxN` pixels.

Precision 1 variant available:

Type	Description
16x8	16-bit signed input, 8-bit unsigned output

Algorithm Algorithm uses ITU-T T.81 (JPEG compression) IDCT-II definition with bias 128 and left-to-right, top-to-bottom orientation.

$$y_{j,i}^{(l)} = 128 + \frac{1}{4}\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} C_m C_n x_{n,m}^{(l)} \cos\frac{(2i+1)m\pi}{2N} \cos\frac{(2j+1)n\pi}{2N}, l = \overline{0...L-1}$$

$$C_k = \begin{cases} 1 & ,k \neq 0 \\ 1/\sqrt{2} & ,k \equiv 0 \end{cases}$$

$x_{n,m}^{(l)} = x[l \cdot N \cdot N + (N \cdot n + m)]$ - sample from n-th row, m-th column from l-th 8x8 block

Prototype

```
int idct2d_16x8(uint8_t * y, int16_t* x, dct_handle_t h, int L, int scalingOpt);
```

2D-IDCT handles:

N	16x8
8	`idct2d_16_8`

`,where N -` IDCT size

Arguments

Type	Name	Size	Description
Input			
`int16_t`	`x`	`N*N*L`	input data: `L NxN` blocks

`dct_handle_t`	`h`		IDCT handle
`int`	`L`		number of input blocks
`int`	`scalingOpt`		scaling option (see table in para 2.9), should be 0
Output			
`uint8_t`	`y`	`N*N*L`	pixels: `L` `NxN` blocks

Returned value 0

Restrictions

`x,y` should not overlap
`x,y` - aligned on 8-bytes boundary

2.10 Identification Routines

2.10.1 Library Version Request

Description This function returns library version information.

Prototype

```
void NatureDSP_Signal_get_library_version(char *version_string);
```

Arguments

Type	Name	Size	Description
Output			
char	version_string	>=30	buffer to store version information

Returned value None

Restrictions version_string must points to a buffer large enough to hold up to 30 characters

Conditions for optimum performance: None

2.10.2 Library API Version Request

Description This function returns library API version information.

Prototype

```
void NatureDSP_Signal_get_library_api_version(char *version_string);
```

Arguments

Type	Name	Size	Description
Output			
char	version_string	>=30	buffer to store version information

Returned value None

Restrictions version_string must point to a buffer large enough to hold up to 30 characters

2.10.3 Library API Capability Request

Description This function returns non-zero if given function (by its address) is supported by specific processor capabilities (i.e., VFPU option).

NOTE:

1. Very few library functions may disable their capabilities dynamically (only for particular combination of input parameters). Behavior for such situation is defined in the description of those functions.

Prototype

```
int NatureDSP_Signal_isPresent(NatureDSP_Signal_funptr fun)
```

Arguments

Type	Name	Description
Input:		
NatureDSP_Signal_funptr	fun	one of library functions

Returned Value non-zero, if function is supported by library

3 Test Environment and Examples

3.1 Supported Use Environment, Configurations and Targets

NatureDSP library and corresponding test suite is supported to be built and test using Xtensa Xplorer IDE running under Windows, or Linux operating system. Also, it might be built:

- by Xtensa tools in command-line mode using makefiles

Library is compatible with HiFi cores having following options:

- HiFi3/HiFi3z base ISA
- HiFi3/HiFi3z Vector FP
- NSA/NSAU ISA option
- MIN/MAX ISA option
- Boolean Registers ISA option
- Little endian target

3.2 Importing the Workspaces in Xtensa Xplorer

NatureDSP Library is provided as two workspaces:

- Library workspace `HiFi3_VFPU_library_v5_0_0.xws`

This workspace contains optimized library kernels

- Test suite workspace `HiFi3_VFPU_demo_v5_0_0.xws`

  This contains the Test Suite application for functional testing and performance measurements.

Import these tow workspaces (`.xws`) in Xtensa Xplorer (XX) as “Xtensa Xplorer workspace”.

Make sure that the library workspace is imported first. This is because the project in the Test_suite workspace has a dependency on the library projects, and the dependency is not correctly set if the library projects are not present when the demo workspace is imported.

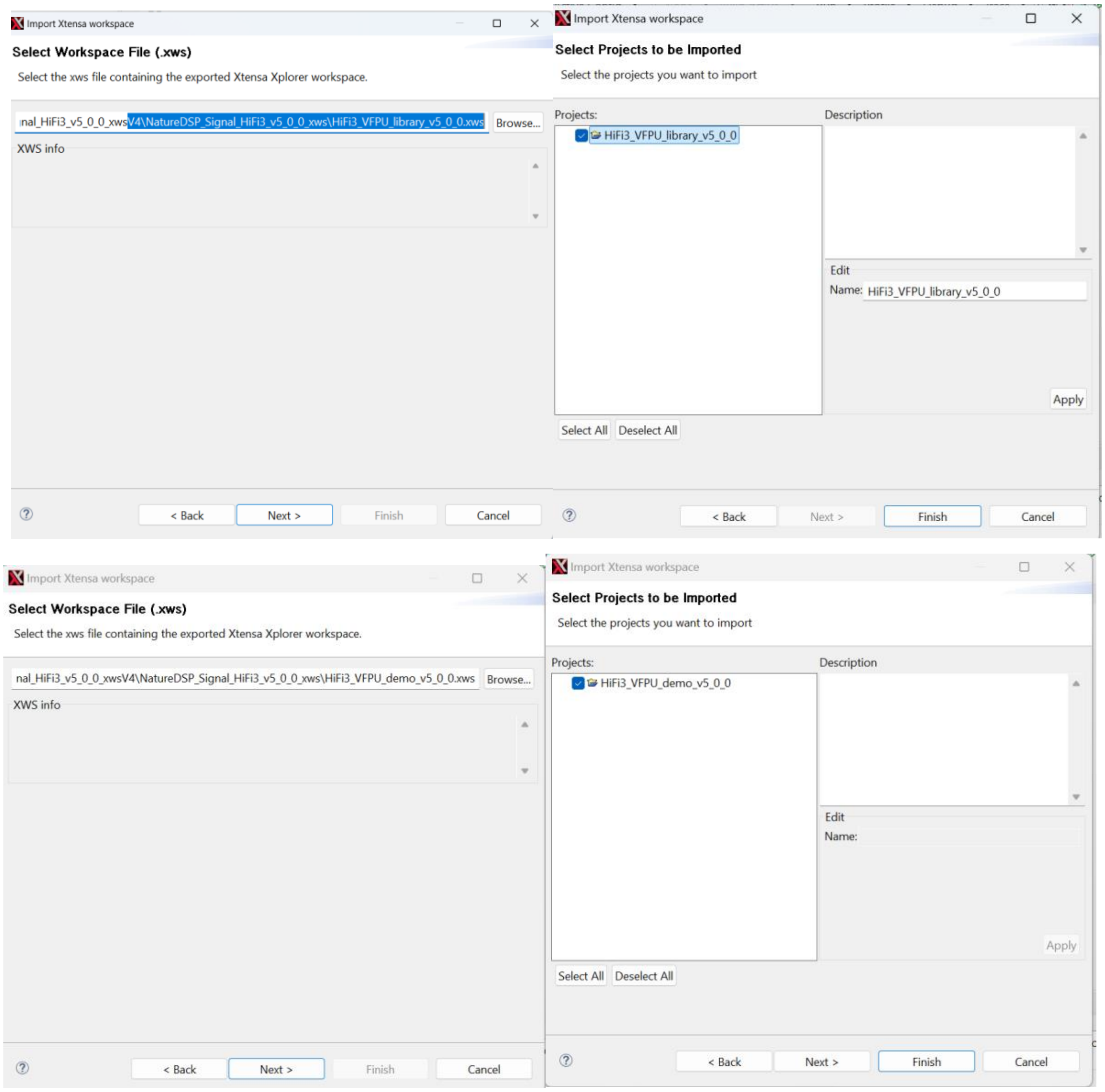
Test vectors are required only for functional tests and not needed for cycle measurements.

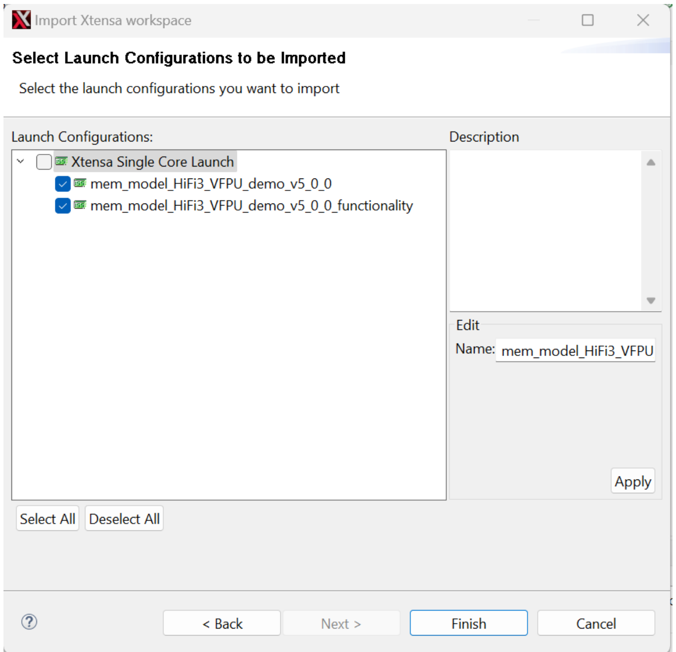
3.2.1 Build and Run NatureDSP Library under XtensaXplorer IDE

To build the library and test suite under XtensaXplorer IDE follow the next steps:

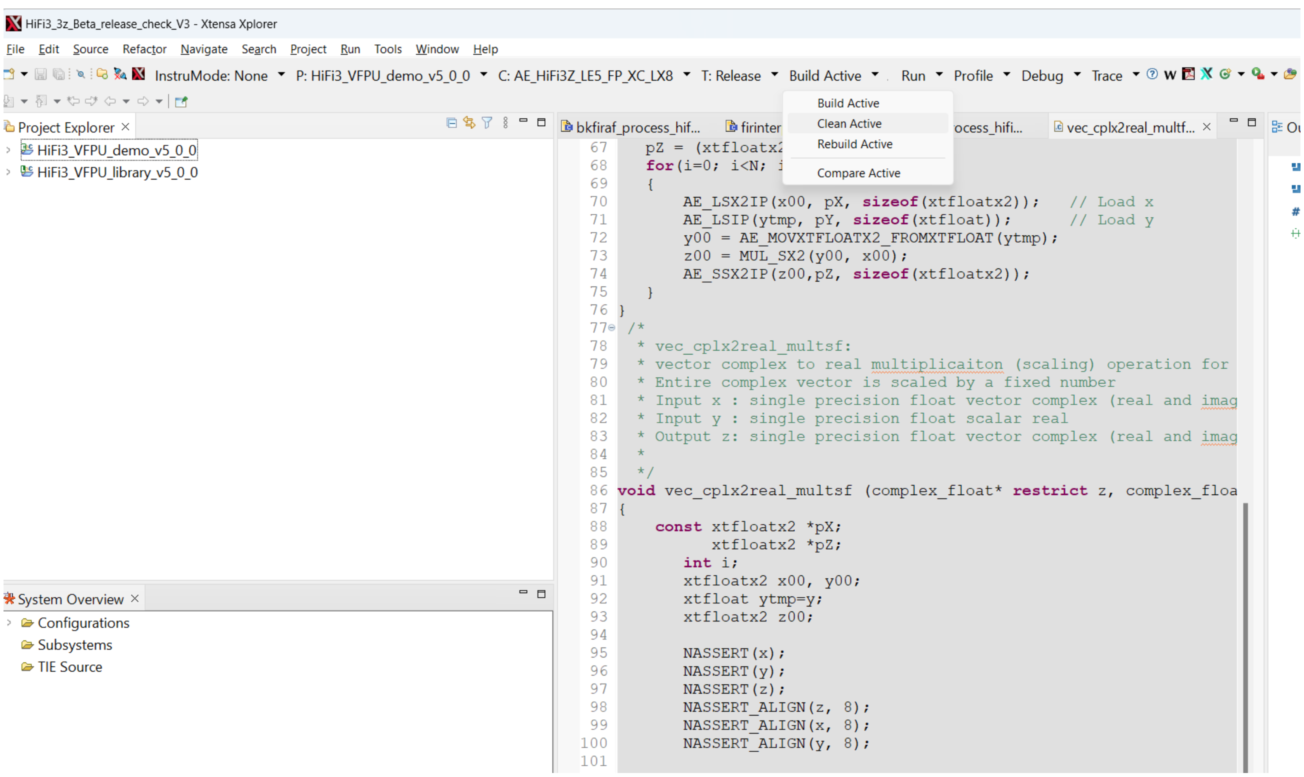
Please note that the core configuration names shown here below are for example, one can use appropriate HiFi 3/3z cores.

1. Import Xtensa Xplorer Workspaces:



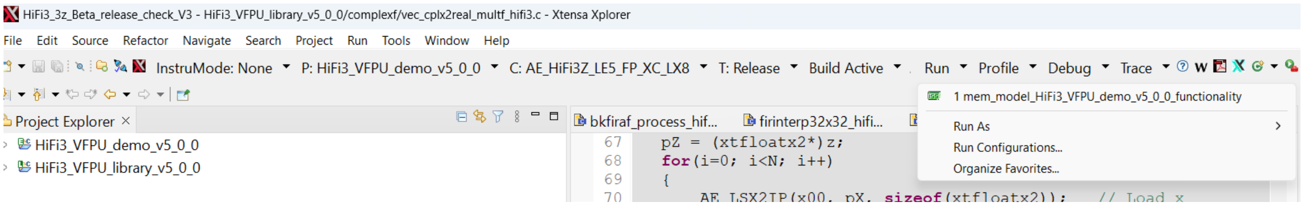


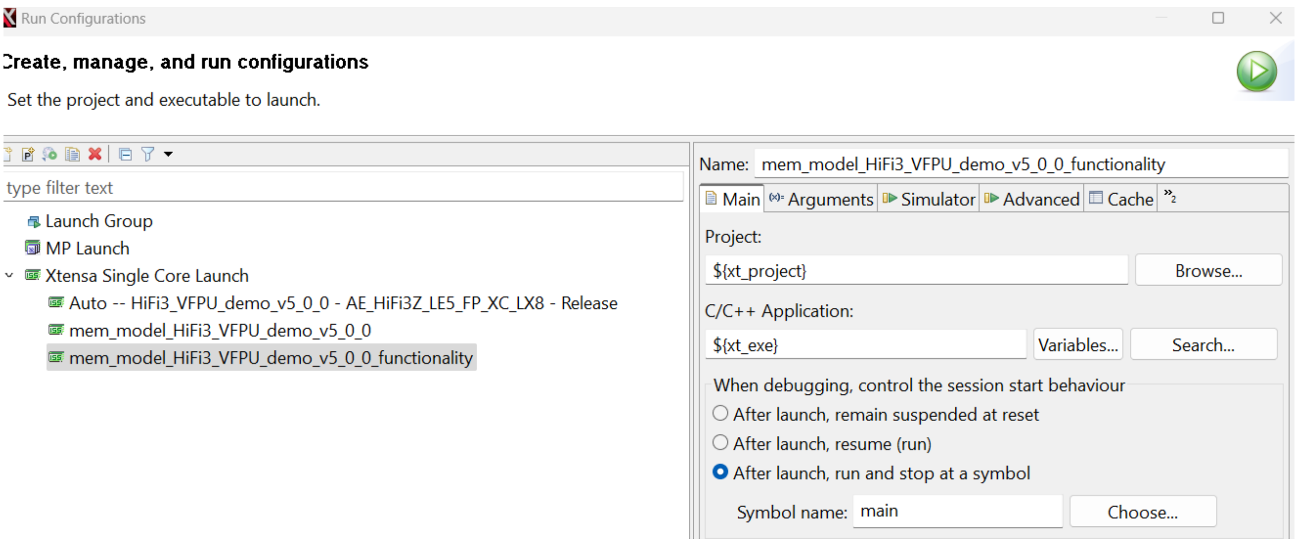
2. Select `HiFi3_VFPU_demo_v5_0_0` as an Active project, compatible core config (`HiFi3Z_LE5_FP_XC` in this example), Active build config (`Release`) and press `Build Active`



For cycle measurement and functional test use Run Configurations `mem_model_HiFi3_VFPU_demo_v5_0_0` and `mem_model_HiFi3_VFPU_demo_v5_0_0_functionality` respectively:

Go to Run>Run Configurations





These configurations use `-mips` for cycle measurements and `-func` for functional tests.

3.2.2 Build and Run the NatureDSP Signal Library with Xtensa Compiler using Command-Line Tools

This kind of build is useful for standalone builds without XtensaXplorer (i.e. when Xtensa tools are accessing via ssh console). It allows to run tests in batch mode for better automation..

For that case, make sure that environment variables `XTENSA_SYSTEM`, `XTENSA_CORE` point to the selected Xtensa Configuration and system `PATH` sets properly to provide an access to Xtensa tools.

To build the NatureDSP Signal Library and the Test Suite

1. Open command shell.
2. Go to directory with library makefile:

```
cd (package root)\NatureDSP_HiFi3_TestSuite\build\project\xtclang\testdriver
```

3. Run `make` with required options

```
make <options>
```

or

```
xt-make <options>
```

if you are running toolchain under Windows OS

Test suite executable allows testing of NatureDSP Signal Library algorithms and intended both for functional testing and cycle measurements. Executable is placed in the directory

```
(package root)\NatureDSP_HiFi3_TestSuite\build\bin
```

Now, the project is ready for build and run. Use `-func` command line argument for functional tests. Note, that the cycle measurements (`-mips`) are not possible with instruction set simulation under xtclang. See para 3.2.3 for the full lists of program arguments.

For running the test suite, enter this directory and call instructions set simulator (ISS) in command-line mode:

```
xt-run <simopt> testdriver_<config_name> <testdriver_options>
```

Note that `--turbo` option makes cycle measurements inaccurate.

Or, simply run linux gcc executable with embedded instruction set emulation

```
./testdriver_<config_name> <testdriver_options>
```

3.2.3 Command-line Options

You may wish to launch a separate test by passing command-line options to the executable:

You may wish to launch a separate test by passing command-line options to the executable:
Running the Testdriver without options performs functional testing of library. Additionally, it may collect statistics and generate validation report showing the number of calls of each specific library function, amount of data passed to/from, sorts of specific tests performed, etc.
Running performance tests for all library functions or for specific category is controlled by command line option `-mips`. In that case, functional testing is not performed, and validation report will be empty.
Brief performance data are formed with `-mips -verbose`. Detailed performance data are prepared with `-mips -full`.

You may wish to launch a separate test by passing command-line options to the executable:

Package	API	Meaning	Option
		List of available options	`-help or -h`
		Performance test	`-mips`
		Functional tests	`-func`
		Generate validation report and statistics after completion of functional testing	`-vreport`
		test fixed point functions only	`-phase1`
		test floating point functions only	`-phase2`
		For functional tests, this switch instructs to use bigger data vectors from directory `vectors_full` instead of `vectors_brief` (test time might be 3 to 5 times longer). For performance test, it controls amount of tests performed for each library function. With this switch the test tool forms detailed testing using bigger set of function parameters, if not - it just makes brief performance data.	`-full`

Package	API	Meaning	Option
		This switch controls number of tests executed (in contrary with `-full`)	`-brief`
		Verbose reporting. For functional tests, it controls verbosity of test reports (i.e., shows number, type of tests performed and detailed error statistics if some test is failed). For performance tests, it adds textual description of each function under the test to the performance log.	`-verbose`
FIR Filters and Related Functions	2.1	all FIR filters	**`-fir`**
		filtering	`-firblk`
		decimation	`-firdec`
		interpolation	`-firint`
		correlation, convolution, dispreading, LMS	`-firother`
IIR Filters	2.2	all IIR filters	**`-iir`**
		biquad filters	`-iirbq`
		lattice filters	`-iirlt`
Math Functions	2.3	all math functions	**`-math`**
		vectorized math	`-mathv`
		vectorized math (fast variants)	`-mathvf`
		scalar math	`-maths`
Complex Math Functions	2.4	all complex functions	**`-complex`**
		vectorized complex math	`-complexv`
		scalar complex math	`-complexs`
Vector Operations	2.4.2	vector operations tests	**`-vector`**
Matrix Operations	2.5.8	all matrix operations tests	**`-matop`**
Matrix Decomposition and Inversion Functions	2.7	all matrix decomposition and inversion	**`-matinv`**
FFT Routines	2.9	all FFT and DCT	**`-fft`**
		complex FFT	`-cfft`
		real FFT	`-rfft`
		mixed radix complex FFT	`-cnfft`
		mixed radix real FFT	`-rnfft`
		complex FFT with optimized memory	`-cfftie`
		real FFT with optimized memory	`-rfftie`
		DCT	`-dct`
Fitting and Interpolation Routines	2.8	all fitting tests	**`-fit`**
		polynomial fitting	`-pfit`

4 Appendix

4.1 Matlab Code for Conversion of SOS Matrix to Coefficients of IIR Functions

Below is example Matlab code to simplify conversion of SOS+G matrices given from the filter design tools into the format of IIR filtering functions.

4.1.1 bqriir24x24_df1 conversion

```
%--------------------------------------------------
% convert SOS+G to coefficients of IIR filter
% (bqriir24x24_df1 function)
% parameters:
% SOS,G    - SOS matrix and gain vector G
% Fs       - sample rate
% nfft     - FFT length for analisys
% output:
% coef     - vector with coefficients, Q30
% gain     - biquad gains, Q15
% scale    - final scale factor (amount of left shifts)
%--------------------------------------------------
function [coef,gain,scale]=cvtsos_bqriir24x24_df1(SOS,G,Fs,nfft)

sz=size(SOS);
M=sz(1);
f=(0:nfft-1)/nfft*(Fs/2);
tf0=sos2freqz(SOS,G,nfft);

coef=[];
Gtotal=prod(G);
G=ones(1,M+1);

% define gain for each stage to provide maximim of tf for intermediate output <=0.5
for m=1:M
    tf=sos2freqz(SOS(1:m,:),[G(1:m) 1],nfft);
    tfmax=max(abs(tf));
    G(m)=min(1,0.5/tfmax);
    % round to nearest Q7 value
    G(m)=min(127,round(G(m)*128))/128;
end

% define last stage shift
dg=Gtotal/prod(G);
scale=ceil(log2(dg));
% correct coefficient of the last stage
d=pow2(1,scale)/dg;
G(M)=G(M)/d;
% output b,a
coef=SOS; coef(:,4)=[]; coef=reshape(coef.',1,numel(coef));
% and convert coefficients to given format
coef = int32(round(1073741824.*coef));
gain = int16(round(32768.*G(1:M)));
scale= int16(scale);
% check results and plot final filter response
sos=reshape(double(coef),5,M).'/1073741824;
sos=[sos(:,1:3) ones(M,1) sos(:,4:5)];
g=(floor(double(gain)/128)*128)/32768;
g(M+1)=pow2(1,double(scale));
tf=sos2freqz(sos,g,nfft);
plot(f,20*log10(abs(tf)),f,20*log10(abs(tf0))); ylim([-80 0]); title('transfer function, dB'); grid
on;

% convert SOS/G to frequency response
```

```
function [tf]=sos2freqz(SOS,G,nfft)
sz=size(SOS);
M=sz(1);
[b,a]=sos2tf(SOS(1,:),[G(1) G(end)]);
tf=freqz(b,a,nfft);
for m=2:M
    [b,a]=sos2tf(SOS(m,:),[G(m) 1]);
    tf=tf.*freqz(b,a,nfft);
end
```

4.1.2 bqriir16x16_df1, bqriir32x16_df1 conversion

```
%------------------------------------------------
% convert SOS+G to coefficients of IIR filter
% (bqriir32x16_df1 function)
% parameters:
% SOS,G     - SOS matrix and gain vector G
% Fs        - sample rate
% nfft      - FFT length for analisys
% output:
% coef      - vector with coefficients, Q30
% gain      - biquad gains, Q15
% scale     - final scale factor (amount of left shifts)
%------------------------------------------------
function [coef,gain,scale]=cvtsos_bqriir32x16_df1(SOS,G,Fs,nfft)
sz=size(SOS);
M=sz(1);
coef=[];
f=(0:nfft-1)/nfft*(Fs/2);
tf0=sos2freqz(SOS,G,nfft);

Gtotal=prod(G);
G=ones(1,M+1);

% define gain for each stage to provide maximim of tf for intermediate output <=0.5
for m=1:M
    tf=sos2freqz(SOS(1:m,:),[G(1:m) 1],nfft);
    tfmax=max(abs(tf));
    G(m)=min(1,0.5/tfmax);
end

% define last stage shift
dg=Gtotal/prod(G);
scale=ceil(log2(dg));
% correct coefficient of the last stage
d=pow2(1,scale)/dg;
G(M)=G(M)/d;
% output b,a
coef=SOS; coef(:,4)=[]; coef=reshape(coef.',1,numel(coef));
% and convert coefficients to given format
coef = int16(round(16384.*coef));
gain = int16(round(32768.*G(1:M)));
scale= int16(scale);
% check results and plot final filter response
sos=reshape(double(coef),5,M).'/16384;
sos=[sos(:,1:3) ones(M,1) sos(:,4:5)];
g=double(gain)/32768;
g(M+1)=pow2(1,double(scale));
tf=sos2freqz(sos,g,nfft);
plot(f,20*log10(abs(tf)),f,20*log10(abs(tf0))); ylim([-80 0]); title('transfer function, dB'); grid
on;

% convert SOS/G to frequency response
function [tf]=sos2freqz(SOS,G,nfft)
sz=size(SOS);
M=sz(1);
[b,a]=sos2tf(SOS(1,:),[G(1) G(end)]);
tf=freqz(b,a,nfft);
for m=2:M
    [b,a]=sos2tf(SOS(m,:),[G(m) 1]);
```

```
    tf=tf.*freqz(b,a,nfft);
end
```

4.1.3 bqriir24x24_df2 conversion

```
%-------------------------------------------------
% convert SOS+G to coefficients of IIR filter
% (bqriir24x24_df2 function)
% parameters:
% SOS,G    - SOS matrix and gain vector G
% Fs       - sample rate
% nfft     - FFT length for analisys
% output:
% coef     - vector with coefficients, Q30
% gain     - biquad gains, Q15
% scale    - final scale factor (amount of left shifts)
%-------------------------------------------------
function [coef,gain,scale]=cvtsos_bqriir24x24_df2(SOS,G,Fs,nfft)

sz=size(SOS);
M=sz(1);
coef=[];
f=(0:nfft-1)/nfft*(Fs/2);
tf0=sos2freqz(SOS,G,nfft);
Gtotal=prod(G);
G=ones(1,M+1);

% define gain for each stage to provide maximim of tf for
% intermediate outputs <=0.5
% note: for DF2 structure we have to check 2 points:
% b0,b1,b2,a1,a2 and 1,0,0,a1,a2
for m=1:M
    tf=sos2freqz(SOS(1:m,:),[G(1:m) 1],nfft);
    tfmax0=max(abs(tf));
    tf=sos2freqz([SOS(1:m-1,:);1 0 0 1 SOS(m,5:6)] ,[G(1:m) 1],nfft);
    tfmax1=max(abs(tf));
    tfmax = max(tfmax0,tfmax1);
    G(m)=min(1,0.5/tfmax);
    % round to nearest Q7 value
    G(m)=min(127,round(G(m)*128))/128;
end

% define last stage shift
dg=Gtotal/prod(G);
scale=ceil(log2(dg));
% correct coefficient of the last stage
d=pow2(1,scale)/dg;
G(M)=G(M)/d;
% output b,a
coef=SOS; coef(:,4)=[]; coef=reshape(coef.',1,numel(coef));
% and convert coefficients to given format
coef = int32(round(1073741824.*coef));
gain = int16(round(32768.*G(1:M)));
scale= int16(scale);
% check results and plot final filter response
sos=reshape(double(coef),5,M).'/1073741824;
sos=[sos(:,1:3) ones(M,1) sos(:,4:5)];
g=(floor(double(gain)/128)*128)/32768;
g(M+1)=pow2(1,double(scale));
[b,a]=sos2tf(sos,g);
tf=sos2freqz(sos,g,nfft);
plot(f,20*log10(abs(tf)),f,20*log10(abs(tf0))); ylim([-80 0]); title('transfer function, dB'); grid
on;

% convert SOS/G to frequency response
function [tf]=sos2freqz(SOS,G,nfft)
sz=size(SOS);
M=sz(1);
[b,a]=sos2tf(SOS(1,:),[G(1) G(end)]);
tf=freqz(b,a,nfft);
```

```
for m=2:M
    [b,a]=sos2tf(SOS(m,:),[G(m) 1]);
    tf=tf.*freqz(b,a,nfft);
end
```

4.1.4 bqriir16x16_df2, bqriir32x16_df2 conversion

```
%---------------------------------------------------
% convert SOS+G to coefficients of IIR filter
% (bqriir32x16_df2 function)
% parameters:
% SOS,G    - SOS matrix and gain vector G
% Fs       - sample rate
% nfft     - FFT length for analisys
% output:
% coef     - vector with coefficients, Q14
% gain     - biquad gains, Q15
% scale    - final scale factor (amount of left shifts)
%---------------------------------------------------
function [coef,gain,scale]=cvtsos_bqriir32x16_df2(SOS,G,Fs,nfft)

sz=size(SOS);
M=sz(1);
coef=[];
f=(0:nfft-1)/nfft*(Fs/2);
tf0=sos2freqz(SOS,G,nfft);
Gtotal=prod(G);
G=ones(1,M+1);

% define gain for each stage to provide maximim of tf for
% intermediate outputs <=0.5
% note: for DF2 structure we have to check 2 points:
% b0,b1,b2,a1,a2 and 1,0,0,a1,a2
for m=1:M
    tf=sos2freqz(SOS(1:m,:),[G(1:m) 1],nfft);
    tfmax0=max(abs(tf));
    tf=sos2freqz([SOS(1:m-1,:);1 0 0 1 SOS(m,5:6)] ,[G(1:m) 1],nfft);
    tfmax1=max(abs(tf));
    tfmax = max(tfmax0,tfmax1);
    G(m)=min(1,0.5/tfmax);
end

% define last stage shift
dg=Gtotal/prod(G);
scale=ceil(log2(dg));
% correct coefficient of the last stage
d=pow2(1,scale)/dg;
G(M)=G(M)/d;
% output b,a
coef=SOS; coef(:,4)=[]; coef=reshape(coef.',1,numel(coef));
% and convert coefficients to given format
coef = int16(round(16384.*coef));
gain = int16(round(32768.*G(1:M)));
scale= int16(scale);

% check results and plot final filter response
sos=reshape(double(coef),5,M).'/16384;
sos=[sos(:,1:3) ones(M,1) sos(:,4:5)];
g=double(gain)/32768;
g(M+1)=pow2(1,double(scale));
tf=sos2freqz(sos,g,nfft);
plot(f,20*log10(abs(tf)),f,20*log10(abs(tf0))); ylim([-80 0]); title('transfer function, dB'); grid
on;

% convert SOS/G to frequency response
function [tf]=sos2freqz(SOS,G,nfft)
sz=size(SOS);
M=sz(1);
[b,a]=sos2tf(SOS(1,:),[G(1) G(end)]);
tf=freqz(b,a,nfft);
```

```
for m=2:M
    [b,a]=sos2tf(SOS(m,:),[G(m) 1]);
    tf=tf.*freqz(b,a,nfft);
end
```

4.1.5 bqriir32x32_df1 conversion

```
%-------------------------------------------------
% convert SOS+G to coefficients of IIR filter
% (bqriir32x32_df1 function)
% parameters:
% SOS,G     - SOS matrix and gain vector G
% Fs        - sample rate
% nfft      - FFT length for analisys
% output:
% coef      - vector with coefficients, Q30
% gain      - biquad gains, Q30
% scale     - final scale factor (amount of left shifts)
%-------------------------------------------------
function [coef,gain,scale]=cvtsos_bqriir32x32_df1(SOS,G,Fs,nfft)

sz=size(SOS);
M=sz(1);
coef=[];
f=(0:nfft-1)/nfft*(Fs/2);
tf0=sos2freqz(SOS,G,nfft);
Gtotal=prod(G);
G=ones(1,M+1);

% define gain for each stage to provide maximim of tf for intermediate output <=0.5
for m=1:M
    tf=sos2freqz(SOS(1:m,:),[G(1:m) 1],nfft);
    tfmax=max(abs(tf));
    G(m)=min(1,0.5/tfmax);
end

% define last stage shift
dg=Gtotal/prod(G);
scale=ceil(log2(dg));
% correct coefficient of the last stage
d=pow2(1,scale)/dg;
G(M)=G(M)/d;
% output b,a
coef=SOS; coef(:,4)=[]; coef=reshape(coef.',1,numel(coef));
% and convert coefficients to given format
coef = int32(round(1073741824.*coef));
gain = int16(round(32768.*G(1:M)));
scale= int16(scale);
% check results and plot final filter response
sos=reshape(double(coef),5,M).'/1073741824;
sos=[sos(:,1:3) ones(M,1) sos(:,4:5)];
g=double(gain)/32768;
g(M+1)=pow2(1,double(scale));
tf=sos2freqz(sos,g,nfft);
plot(f,20*log10(abs(tf)),f,20*log10(abs(tf0))); ylim([-80 0]); title('transfer function, dB'); grid
on;

% convert SOS/G to frequency response
function [tf]=sos2freqz(SOS,G,nfft)
sz=size(SOS);
M=sz(1);
[b,a]=sos2tf(SOS(1,:),[G(1) G(end)]);
tf=freqz(b,a,nfft);
for m=2:M
    [b,a]=sos2tf(SOS(m,:),[G(m) 1]);
    tf=tf.*freqz(b,a,nfft);
end
```

4.1.6 bqriir32x32_df2 conversion

```
%--------------------------------------------------
% convert SOS+G to coefficients of IIR filter
% (bqriir32x32_df2 function)
% parameters:
% SOS,G    - SOS matrix and gain vector G
% Fs       - sample rate
% nfft     - FFT length for analisys
% output:
% coef     - vector with coefficients, Q30
% gain     - biquad gains, Q15
% scale    - final scale factor (amount of left shifts)
%--------------------------------------------------
function [coef,gain,scale]=cvtsos_bqriir32x32_df2(SOS,G,Fs,nfft)

sz=size(SOS);
M=sz(1);
coef=[];
f=(0:nfft-1)/nfft*(Fs/2);
tf0=sos2freqz(SOS,G,nfft);
Gtotal=prod(G);
G=ones(1,M+1);

% define gain for each stage to provide maximim of tf for
% intermediate outputs <=0.5
% note: for DF2 structure we have to check 2 points:
% b0,b1,b2,a1,a2 and 1,0,0,a1,a2
for m=1:M
    tf=sos2freqz(SOS(1:m,:),[G(1:m) 1],nfft);
    tfmax0=max(abs(tf));
    tf=sos2freqz([SOS(1:m-1,:);1 0 0 1 SOS(m,5:6)] ,[G(1:m) 1],nfft);
    tfmax1=max(abs(tf));
    tfmax = max(tfmax0,tfmax1);
    G(m)=min(1,0.5/tfmax);
end

% define last stage shift
dg=Gtotal/prod(G);
scale=ceil(log2(dg));
% correct coefficient of the last stage
d=pow2(1,scale)/dg;
G(M)=G(M)/d;
% output b,a
coef=SOS; coef(:,4)=[]; coef=reshape(coef.',1,numel(coef));
% and convert coefficients to given format
coef = int32(round(1073741824.*coef));
gain = int16(round(32768.*G(1:M)));
scale= int16(scale);

% check results and plot final filter response
sos=reshape(double(coef),5,M).'/1073741824;
sos=[sos(:,1:3) ones(M,1) sos(:,4:5)];
g=double(gain)/32768;
g(M+1)=pow2(1,double(scale));
tf=sos2freqz(sos,g,nfft);
plot(f,20*log10(abs(tf)),f,20*log10(abs(tf0))); ylim([-80 0]); title('transfer function, dB'); grid
on;

% convert SOS/G to frequency response
function [tf]=sos2freqz(SOS,G,nfft)
sz=size(SOS);
M=sz(1);
[b,a]=sos2tf(SOS(1,:),[G(1) G(end)]);
tf=freqz(b,a,nfft);
for m=2:M
    [b,a]=sos2tf(SOS(m,:),[G(m) 1]);
    tf=tf.*freqz(b,a,nfft);
end
```

4.1.7 bqriirf_df1, bqriirf_df2, bqriirf_df2t conversion

```
%--------------------------------------------------
% convert SOS+G coefficients to the coefficients for
% iirdf1f,iirdf2f,iirdf2tf,ciirdf1f routines
% parameters:
% SOS,G    - SOS matrix and gain vector G
% Fs       - sample rate
% nfft     - FFT length for analisys
% output:
% coef     - vector with coefficients
%--------------------------------------------------
function [coef,scale]=cvtsos_iir(SOS,G,Fs,nfft)
% convert SOS+G to coefficients of IIR filter
sz=size(SOS);
M=sz(1);
coef=[];
f=(0:nfft-1)/nfft*(Fs/2);
tf0=sos2freqz(SOS,G,nfft);
Gtotal=prod(G);
G=ones(1,M+1);

% define gain for each stage to provide maximim of tf for intermediate output <=0.5
for m=1:M
    tf=sos2freqz(SOS(1:m,:),[G(1:m) 1],nfft);
    tfmax=max(abs(tf));
    G(m)=min(1,0.5/tfmax);
end
for m=1:M
    SOS(m,1:3)= SOS(m,1:3)*G(m);
end
% define last stage shift
dg=Gtotal/prod(G);
scale=round(log2(dg));
% correct coefficient of the last stage
d=pow2(1,scale)/dg;
G(M)=G(M)/d;
% output b,a
coef=SOS'; coef(4,:)=[]; coef=reshape(coef,1,numel(coef));
scale= int32(scale);

% check results and plot final filter response
sos=reshape(double(coef),5,M);
sos=sos';
sos=[sos(:,1:3) ones(M,1) sos(:,4:5)];
g(1:M)=ones(1,M);
g(M+1)=pow2(1,double(scale));
tf=sos2freqz(sos,g,nfft);
plot(f,20*log10(abs(tf)),f,20*log10(abs(tf0))); ylim([-80 0]); title('transfer function, dB'); grid
on;

% convert SOS/G to frequency response
function [tf]=sos2freqz(SOS,G,nfft)
sz=size(SOS);
M=sz(1);
[b,a]=sos2tf(SOS(1,:),[G(1) G(end)]);
tf=freqz(b,a,nfft);
for m=2:M
    [b,a]=sos2tf(SOS(m,:),[G(m) 1]);
    tf=tf.*freqz(b,a,nfft);
end
```

4.2 Matlab Code for Generation the Twiddle Tables

FFT with optimized memory usage require external twiddle tables. Matlab code below shows how to generate twiddles for different functions.

4.2.1 Twiddles for fft_cplx24x24_ie, ifft_cplx24x24_ie, fft_real24x24_ie, ifft_real24x24_ie

```
function [twd]=twd24x24_ie(N)
twd = exp(-2j*pi*[1;2;3]*(0:N/4-1)/N);
twd=twd.';
twd = reshape([real(twd(:).');imag(twd(:).')],1,2*numel(twd));
twd = int32(round(pow2(twd,31)));
```

4.2.2 Twiddles for fft_cplx32x16_ie, ifft_cplx32x16_ie, fft_real32x16_ie, ifft_real32x16_ie, fft_cplx16x16_ie, ifft_cplx16x16_ie, fft_real16x16_ie, ifft_real16x16_ie

```
function [twd]=twd32x16_ie(N)
twd = exp(-2j*pi*[1;2;3]*(0:N/4-1)/N);
twd=twd.';
twd = reshape([imag(twd(:).');real(twd(:).')],1,2*numel(twd));
twd = int16(round(pow2(twd,15)));
```

4.2.3 Twiddles for fft_cplx32x32_ie, ifft_cplx32x32_ie, fft_real32x32_ie, ifft_real32x32_ie

```
function [twd]=twd32x32_ie(N)
twd = exp(-2j*pi*[1;2;3]*(0:N/4-1)/N);
twd=twd.';
twd = reshape([imag(twd(:).');real(twd(:).')],1,2*numel(twd));
twd = int32(round(pow2(twd,31)));
```

4.2.4 Twiddles for fft_cplxf_ie, ifft_cplxf_ie, fft_realf_ie, ifft_realf_ie

```
function [twd]=twdf_ie(N)
twd = exp(-2j*pi*[1;2;3]*(0:N/4-1)/N);
twd = reshape([real(twd(:).');imag(twd(:).')],1,2*numel(twd));
```