NatureDSP Signal Library for HiFi 1 DSP

Digital Signal Processing

Library Reference

Library Release 1.1.0
API Revision 1.1
Oct, 2022

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

IMPORTANT NOTICE

Table of Contents

Document History

Revision	Date	Major changes
1.0	16 Nov 2021	First release version.
1.1	10 Oct 2022	Second release version

Preface

About This Manual

Welcome to the **NatureDSP Signal Processing Library**, or **NatureDSP Signal** or library for short. The library is a collection of a number of highly optimized DSP functions for the HiFi 1 DSP.

This source code library includes C-callable functions (ANSI-C language compatible) for general signal processing (filtering, correlation, convolution), math and vector functions specially optimized for high-density signal processing. Library supports both fixed-point and single precision floating data types. These library APIs are developed based on Nature DSP APIs developed by IntegrIT.

Supported Targets

Library supports Cadence Tensilica HiFi 1 DSP with VFPU option. HiFi 1 DSP supports only little-endian configuration. RI-2022.9 toolchain shall be used for building HiFi1e extension cores with enhanced ISA. The library still supports backward compatibility for cores with earlier toolchain.

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
LSB	Least significant bit
ULP	Units of least precision
VFPU	Vector Floating Point Unit
XX	Xtensa Xplorer

1 General Library Organization

1.1 Headers

NatureDSP_Signal library is supplied with following header files

`./include/NatureDSP_types.h`	Declarations of basic data types and compiler auto detection
`./include/NatureDSP_Signal.h`	Declarations of library functions

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
`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
`f24`	24-bit fractional type	4

It is assumed throughout this reference manual that constant pointers passed through function arguments point at read-only data.
Normally, `f24` fractional data are stored at 3 higher bytes of 32-bit words and 8 LSBs are assumed to be 0. However, a few routines use packed 24-bit data where 24-bit fractional numbers are allocated only 3 consecutive bytes.
Data of given type should be aligned on its `sizeof()`, see table above.

1.4 Fractional Formats

Natively, HiFi 1 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 assumed to be 0s. **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 `Qm`.

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
int16_t	Q3.12	-8 … 7,9998	2e-4	-32768	32767
int16_t	Q8.7	-256 … 255,992	8e-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.
For example, if a number is representing in Q31 format and have decimal representation N, its floating-point equivalent will be N^2-31.

1.5 Compiler Requirements

When building the library source files or library-dependent modules it is assumed that the target is a Tensilica processor implementing the Xtensa HiFi 1 Instruction Set Architecture with or without VFPU option. This is accomplished through checking the `COMPILER_XTENSA` preprocessor symbol performed in `NatureDSP_types.h`. This symbol is automatically defined by the compiler. The library is delivered as an Xtensa Xplorer workspace, along with a second workspace containing demonstrations and tests of the various library routines. The library is intended to be used with Xtensa Xplorer. When the workspaces are imported, all relevant compiler options are automatically set, although the user can modify them if needed.

1.6 Call Conventions

Library uses ANSI-C call conventions.

1.7 Overflow Control and Intermediate Data Format

If not specially noted, library does not check the real dynamic range of input data, so it is user's responsibility to select parameters and the scale of input data according to each specific case. Wherever possible library uses saturated arithmetic to prevent overflows.
In most of the fixed-point routines with operations involving 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 of restricting 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 VFPU enabled bits. VFPU 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. The end implementation is expected 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 Endianness

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, users 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.
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 slightly bigger size and align pointers[1]

[1] It shall be the application's responsibility to pass the pointers with required alignment.

Test examples included, uses the last two methods.

1.12 Object Model

Effective use of all HiFi 1 core benefits requires 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 HiFi 1 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 allocated for to the 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 the object.

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
firinterp		Interpolating block real FIR filter	2.1.5
fir_convol, cxfir_convol		Circular convolution	2.1.6
fir_lconvol		Linear convolution	2.1.7
fir_xcorr		Circular correlation	2.1.8
fir_lxcorr		Linear correlation	2.1.9
fir_acorr		Circular autocorrelation	2.1.9
fir_lacorr		Linear autocorrelation	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
Vector mathematics			
vec_dot		Vector dot product	2.3.1
vec_add		Vector sum	2.3.2
vec_power		Power of a vector	2.3.3
vec_shift vec_scale		Vector scaling with saturation	2.3.4
vec_recip	scl_recip	Reciprocal	2.3.5

Vectorized version	Scalar version	Purpose	Reference
vec_divide	scl_divide	Division	2.3.6
vec_logn	scl_logn	Different kinds of logarithm	2.3.7
vec_log2	scl_log2		
vec_logn	scl_logn		
vec_antilog2	scl_antilog2	Different kinds of antilogarithm	2.3.8
vec_antilog10	scl_antilog10		
vec_antilogn	scl_antilogn		
vec_sqrt	scl_sqrt	Square root	2.3.9
vec_rsqrt	scl_rsqrt	Reciprocal square root	2.3.10
vec_sine	scl_sine	Sine	2.3.11
vec_cosine	scl_cosine	Cosine	
vec_tan	scl_tan	Tangent	2.3.12
vec_asin	scl_asin	Arcsine	2.3.13
vec_acos	scl_acos	Arccosine	2.3.13
vec_atan, vec_atan2	scl_atan, scl_atan2	Arctangent	2.3.14, 2.3.15
vec_bexp	scl_bexp	Common exponent	2.3.16
vec_min, vec_max		Find a maximum/minimum in a vector	2.3.17
vec_poly		Polynomial approximation	2.3.18
vec_int2float	scl_int2float	Integer to float conversion	2.3.19
vec_float2int	scl_float2int	Float to integer conversion	2.3.20
vec_float2floor	scl_float2floor	Rounding	2.3.21
vec_float2ceil	scl_float2ceil		
vec_complex2mag	scl_complex2mag	Complex magnitude	2.3.22
vec_complex2invmag	scl_complex2invmag	Reciprocal of complex magnitude	2.3.22
Matrix operations			
mtx_mpy		Matrix multiply	2.4.1
mtx_vecmpy		Matrix by vector multiple	2.4.2
mtx_add, cmtx_add		Matrix addition	2.4.3
mtx_sub, cmtx_sub		Matrix subtraction	2.4.3
mtx_mul, cmtx_mul		Matrix multiply	2.4.3
mtx_tran, cmtx_tran		Matrix transpose	2.4.3
mtx_det, cmtx_det		Matrix determinant	2.4.3
mtx_inv, cmtx_inv		Matrix inverse	2.4.4
q2rot		Quaternion to Rotation Matrix Conversion	2.4.5
FFT			
fft_cplx		FFT on complex data	2.5.1
fft_real		FFT on real data	2.5.2
ifft_cplx		Inverse FFT on complex data	2.5.3
ifft_real		Inverse FFT forming real data	2.5.4
dct		Discrete cosine transform	2.5.5
fft_cplx_ie		FFT on complex data with optimized memory usage	2.5.6
fft_real_ie		FFT on real data with optimized memory usage	2.5.7
ifft_cplx_ie		Inverse FFT on complex data with optimized memory usage	2.5.8
ifft_real_ie		Inverse FFT forming real data with optimized memory usage	2.5.9
Identification			
NatureDSP_Signal_get_library_version		Library Version Request	2.6.1
NatureDSP_Signal_get_library_api_version		Library API Version Request	2.6.2

2 Reference

2.1 FIR Filters and Related Functions

FIR filtering APIs excepting correlation/convolution, autocorrelation and block wise LMS algorithm require instantiation. In particular, filter objects encapsulates 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.

Precision

6 versions available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit outputs
24x24	24-bit data, 24-bit coefficients, 24-bit outputs
24x24p	24-bit data, 24-bit coefficients, 24-bit outputs
32x16	32-bit data, 16-bit coefficients, 32-bit outputs
32x32	32-bit data, 32-bit coefficients, 32-bit outputs
f	floating point

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)
size_t bkfir24x24_alloc (int M)
size_t bkfir24x24p_alloc(int M)
size_t bkfir32x16_alloc (int M)
size_t bkfir32x32_alloc (int M)
size_t bkfirf_alloc (int M)
```

Type	Name	Size	Description
Input			
`int`	`M`		length of filter, should be a multiple of 4

Returns: size of memory in bytes to be allocated

NOTE:
Approximate amount of requested memory is listed below

Function	Approximate memory requirements, bytes
`bkfir16x16_alloc`	`72+M*4`
`bkfir24x24_alloc`	`72+M*8`

bkfir24x24p_alloc	72+M*8
bkfir32x16_alloc	72+M*6
bkfir32x32_alloc	72+M*8
bkfirf_alloc	72+M*8

Object initialization

```
bkfir16x16_handle_t bkfir16x16_init
           (void * objmem, int M, const int16_t *  h)
bkfir24x24_handle_t bkfir24x24_init
           (void * objmem, int M, const f24 *  h)
bkfir24x24p_handle_t bkfir24x24p_init
           (void * objmem, int M, const f24 *  h)
bkfir32x16_handle_t bkfir32x16_init
           (void * objmem, int M, const int16_t*  h)
bkfir32x32_handle_t bkfir32x32_init
           (void * objmem, int M, const int32_t*  h)
bkfirf_handle_t bkfirf_init
           (void * objmem, int M, 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

Returns: handle to the object

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 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,handle` - aligned on 8-byte 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

5 versions available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit outputs
24x24	24-bit data, 24-bit coefficients, 24-bit outputs
32x16	32-bit data, 16-bit coefficients, 32-bit outputs
32x32	32-bit data, 32-bit coefficients, 32-bit outputs
f	floating point

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 but internally 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 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`
`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)
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 *  x, 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).

Precision

5 versions available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit outputs
24x24	24-bit data, 24-bit coefficients, 24-bit outputs
32x16	32-bit data, 16-bit coefficients, 32-bit outputs
32x32	32-bit data, 32-bit coefficients, 32-bit outputs
f	floating point

Algorithm

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

NOTE:
This is formal description of algorithm, but internally 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)
size_t cxfir24x24_alloc(int M)
size_t cxfir32x16_alloc(int M)
size_t cxfir32x32_alloc(int M)
size_t cxfirf_alloc(int M)
```

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
`cxfir16x16_alloc`	`80+8*M`
`cxfir32x16_alloc`	`80+12*M`
`cxfir24x24_alloc`	`64+16*M`
`cxfir32x32_alloc`	`64+16*M`
`cxfirf_alloc`	`64+16*M`

Object initialization

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

Returns: handle to the object

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 cxfirf (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, handle` - aligned on 8-byte 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

5 versions available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit outputs
24x24	24-bit data, 24-bit coefficients, 24-bit outputs
32x16	32-bit data, 16-bit coefficients, 32-bit outputs
32x32	32-bit data, 32-bit coefficients, 32-bit outputs
f	floating point

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 but internally 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 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
`firdec16x16_alloc`	`40+(M+4*D)*4+(M+4)*2`
`firdec32x16_alloc`	`40+(M+8*D)*4+(M+4)*2`
`firdec24x24_alloc`	`40+(M+8*D)*4+(M+4)*4`
`firdec32x32_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)
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 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, handle` - aligned on 8-byte boundary
`N` – multiple of 8
`D >1`

Conditions for optimum performance

`D` – 2, 3 or 4

2.1.5 Interpolating Block Real/Complex FIR Filter

Description

Computes a real FIR filter (direct-form) with interpolation using IR stored in vector `h`. The real/complex 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 versions available:

Type	Description
16x16	16-bit real data, 16-bit coefficients, 16-bit real outputs
16x16	16-bit complex data, 16-bit coefficients, 16-bit complex outputs
24x24	24-bit real data, 24-bit coefficients, 24-bit real outputs
32x16	32-bit real data, 16-bit coefficients, 32-bit real outputs
32x32	32-bit real data, 32-bit coefficients, 32-bit real outputs
f	floating point

Algorithm

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

NOTE:
This is formal description of algorithm but internally 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 cxfirinterp16x16_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 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`
`cxfirinterp16x16_alloc`	`40+(M+8)*4+(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`
`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)
cxfirinterp16x16_handle_t cfirinterp16x16_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)
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 cxfirinterp16x16_process(cfirinterp16x16_handle_t handle,
                              complex_fract16 *  y, const complex_fract16 *  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 firinterpf_process     (firinterpf_handle_t handle,
                              float32_t * y, const float32_t * x, int N);
```

Type	Name	Size	Description
Input			
`int16_t, complex_fract16, f24, int32_t, float32_t`	`x`	`N`	input samples,Q15, Q31 or floating point
`int`	`N`		length of input sample block
Output			
`int16_t, complex_fract16, 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 variants of these functions available: faster version (fir_convol16x16, fir_convol24x24, fir_convol32x16, cxfir_convol32x16, fir_convol32x32, fir_convolf) with some restrictions on input arguments and slower version (fir_convola16x16, fir_convola24x24, fir_convola32x16, cxfir_convola32x16, fir_convola32x32, fir_convolaf) for arbitrary arguments. In addition, these slower version implementations require scratch memory area.

Precision

5 versions available:

Type	Description
16x16	16x16-bit data, 16-bit outputs
24x24	24x24-bit data, 24-bit outputs
32x16	32x16-bit data, 32-bit outputs (both real and complex)
32x32	32x32-bit data, 32-bit outputs
f	floating point

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 cxfir_convol32x16 (complex_fract32 *  r,
                        const complex_fract32 *  x,
                        const complex_fract16 *  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_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 cxfir_convola32x16 (void    *  s,
                         complex_fract32 *  r,
                         const complex_fract32 *  x,
                         const complex_fract16 *  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_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)
int32_t, 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) CXFIR_CONVOLA32X16_SCRATCH_SIZE(N,M) FIR_CONVOLA32X32_SCRATCH_SIZE(N, M) FIR_CONVOLAF_SCRATCH_SIZE(N, M) bytes

Returned value none

Restrictions

For slow versions (`fir_convola16x16, fir_convola24x24, fir_convola32x16, cxfir_convola32x16, fir_convola32x32, fir_convolaf`):
`x,y,r,s` should not overlap
`s` should be aligned on 8-byte boundary
`N>0, M>0`
`N>=M-1`

For fast versions (`fir_convol16x16, fir_convol24x24, fir_convol32x16, cxfir_convol32x16, fir_convol32x32, fir_convolf`):
`x,y,r` should not overlap
`x,y,r` should be aligned on 8-byte boundary
`N>0, M>0`
`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

4 versions available:

Type	Description
16x16	16x16-bit data, 16-bit outputs
32x16	32x16-bit data, 32-bit outputs
32x32	32x32-bit data, 32-bit outputs
f	floating point

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_lconvola32x16 (void      *  s,
                         int32_t  *  r,
                         const int32_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);

void fir_lconvolaf     (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, int32_t, float32_t`	`x`	`N`	input data (Q15, Q31 or floating point)
`int16_t, int32_t, float32_t`	`y`	`M`	input data (Q31, Q15 or floating point)
`int`	`N`		length of `x`
`int`	`M`		length of `y`
Output			
`int16_t, int32_t, float32_t`	`r`	`M+N-1`	output data, Q15, Q31 or floating point
Temporary			
`void`	`s`		Scratch memory, `FIR_LCONVOLA16X16_SCRATCH_SIZE( N, M)` `FIR_LCONVOLA32X16_SCRATCH_SIZE( N, M)` `FIR_LCONVOLA32X32_SCRATCH_SIZE( N, M)` `FIR_LCONVOLAF_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 similar to convolution but `y` is read in opposite direction.
Two variants of these functions available: faster version (`fir_xcorr16x16, fir_xcorr24x24, fir_xcorr32x16, fir_xcorr32x32, fir_xcorrf, cxfir_xcorrf`) with some restrictions on input arguments and slower version (`fir_xcorra16x16, fir_xcorra24x24, fir_xcorra32x16, fir_xcorr32x32, fir_xcorraf, cxfir_xcorraf`) for arbitrary arguments. In addition, these slower version implementations require scratch memory area.

Precision

5 versions available:

Type	Description
16x16	16x16-bit data, 16-bit outputs
24x24	24x24-bit data, 24-bit outputs
32x16	32x16-bit data, 32-bit outputs
32x32	32x32-bit data, 32-bit outputs
f	floating point (both real and complex data)

Algorithm

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

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, int M);

void fir_xcorr32x32 (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_xcorra16x16 (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_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);
```

```
void fir_xcorra16x16 (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_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

Type	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_XCORRA32X32_SCRATCH_SIZE(N, M) FIR_XCORRAF_SCRATCH_SIZE(N, M) CXFIR_XCORRAF_SCRATCH_SIZE(N, M) FIR_XCORRA32X16_SCRATCH_SIZE(N, M) bytes

Returned value

none

Restrictions

For slow versions (`fir_xcorra16x16, fir_xcorra24x24, fir_xcorra32x16, fir_xcorra32x32, fir_xcorraf, cxfir_xcorraf`):
`x,y,r,s` should not overlap
`s` should be aligned on 8-byte boundary
`N>0, M>0`
`N>=M-1`

For fast versions (`fir_xcorr16x16, fir_xcorr24x24, fir_xcorr32x16, fir_xcorr32x32, fir_xcorrf, cxfir_xcorrf`):
`x,y,r` should not overlap
`x,y,r` should be aligned on 8-byte boundary
`N>0, M>0`
`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

4 versions available:

Type	Description
16x16	16x16-bit data, 16-bit outputs
32x16	32x16-bit data, 32-bit outputs
32x32	32x32-bit data, 32-bit outputs
f	floating point

Algorithm

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

Prototype

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

void fir_lxcorra32x16 (void     *  s,
                       int32_t   *  r,
                       const int32_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);

void fir_lxcorraf      (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, int32_t, float32_t`	`x`	`N`	input data (Q15, Q31 or floating point)
`int16_t, int32_t, float32_t,`	`y`	`M`	input data (Q31, Q15 or floating point)
`int`	`N`		length of `x`
`int`	`M`		length of `y`
Output			
`int16_t, int32_t, float32_t`	`r`	`M+N-1`	output data, Q15, Q31 or floating point
Temporary			
`void`	`s`		Scratch memory, `FIR_LXCORRA16X16_SCRATCH_SIZE(N, M)` `FIR_LXCORRA32X16_SCRATCH_SIZE(N, M)` `FIR_LXCORRA32X32_SCRATCH_SIZE(N, M)` `FIR_LXCORRAF_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 variants of these functions available: faster version (`fir_acorr24x24, fir_acorrf`) with some restrictions on input arguments and slower version (`fir_acorra24x24, fir_acorraf`) for arbitrary arguments. In addition, this slower version implementations require scratch memory area.

Precision

4 versions available:

Type	Description
16x16	16-bit data, 16-bit outputs
24x24	24-bit data, 24-bit outputs
32x32	32-bit data, 32-bit outputs
f	floating point

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_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_acorraf (void* s,
                  float32_t*  r, const float32_t*  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int16_t, f24, int32_t or float32_t`	`x`	`N`	input data (Q15, Q31 or floating point)
`int`	`N`		length of `x`
Output			
`int16_t, f24, int32_t 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_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_acorraf`):
`x,r` should not overlap
`x,r` should be aligned on 8-byte boundary
`N>0`
`N` –multiple of 4

2.1.11 Linear Autocorrelation

Description

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

Precision

3 versions available:

Type	Description
16x16	16-bit data, 16-bit outputs
32x32	32-bit data, 32-bit outputs
f	floating point

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);
void fir_lacorraf     (void* s, float32_t*  r, const float32_t*  x, int N);
```

Arguments

Type	Name	Size	Description
Input			
`int16_t, int32_t or float32_t`	`x`	`N`	input data (Q15, Q31 or floating point)
`int`	`N`		length of `x`
Output			
`int16_t, int32_t or float32_t`	`r`	`N`	output data, Q15, Q31 or floating point
Temporary			
`void`	`s`		Scratch memory, `FIR_LACORRA16X16_SCRATCH_SIZE( N )` `FIR_LACORRA32X32_SCRATCH_SIZE( N )` `FIR_LACORRAF_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 input samples `x[N+M-1]`, computation of error `e[N]` over a block of reference data `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 input signal times `N` - the number of samples in a data block. This can be calculated i.e. by using the `vec_power24x24()` or `vec_power16x16()` functions. 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

5 versions available:

Type	Description
16x16	16-bit coefficients, 16-bit data, 16-bit output
24x24	24-bit coefficients, 24-bit data, 32-bit output
16x32	32-bit coefficients, 16-bit data, 16-bit output
32x32	32-bit coefficients, 32-bit data, 32-bit output
f	floating point

Algorithm

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

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

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

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, Q15, Q31 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, Q15, Q31

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 Block IIR

Description

Computes a 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.

Filters are able to process data in following formats:

- real (just array of samples)
- 2-way or complex (interleaved real/imaginary samples)
- 3-way (stream of interleaved samples from 3 channels)

The same coefficients are used for filtering of multiple channels or real/imaginary parts and they are processed independently.

The same format have to be used both for input and output streams.

NOTES:

1. 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.
2. 16x16 filters may suffer more from accumulation of the roundoff errors, so filters should be properly designed to match noise requirements

Precision

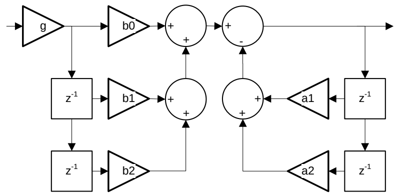
10 versions available:

Type	Description
16x16	16-bit data, 16-bit coefficients, 16-bit intermediate stage outputs (DF I, DF II), real data
16x16	16-bit data, 16-bit coefficients, 16-bit intermediate stage outputs (DF I, DF II), 3-way data
24x24	32-bit data, 24-bit coefficients, 32-bit intermediate stage outputs (DF I, DF II), real data
32x16	32-bit data, 16-bit coefficients, 32-bit intermediate stage outputs (DF I, DF II), real data
32x16	32-bit data, 16-bit coefficients, 32-bit intermediate stage outputs (DF I, DF II), 3-way data
32x32	32-bit data, 32-bit coefficients, 32-bit intermediate stage outputs (DF I, DF II)
32x32	32-bit data, 32-bit coefficients, 32-bit intermediate stage outputs (DF I, DF II) 3-way data
f	floating point (DF I, DF II and DF IIt)
f	floating point (DF I), 2-way (complex) data
f	floating point (DF I, DF II) 3-way data

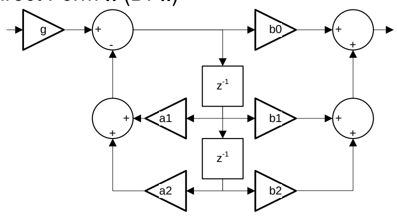
Algorithm

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:

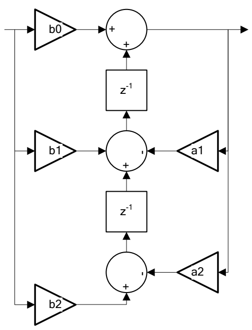
Direct Form I (DFI)



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 bq3iir16x16_df1_alloc(int M)
size_t bq3iir16x16_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 bq3iir32x16_df1_alloc(int M)
size_t bq3iir32x16_df2_alloc(int M)
size_t bqriir32x32_df1_alloc(int M)
size_t bqriir32x32_df2_alloc(int M)
size_t bq3iir32x32_df1_alloc(int M)
size_t bq3iir32x32_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)
size_t bq3iirf_df1_alloc(int M)
size_t bq3iirf_df2_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);
bq3iir16x16_df1_handle_t bq3iir16x16_df1_init(void * objmem, int M,
            const int16_t * coef_sos, const int16_t * coef_g, int16_t gain );
bq3iir16x16_df2_handle_t bq3iir16x16_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);
bq3iir32x16_df1_handle_t bq3iir32x16_df1_init(void * objmem, int M,
            const int16_t * coef_sos, const int16_t * coef_g, int16_t gain);
bq3iir32x16_df2_handle_t bq3iir32x16_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)
bq3iir32x32_df1_handle_t bq3iir32x32_df1_init(void * objmem, int M,
            const int32_t * coef_sos, const int16_t * coef_g, int16_t gain)
bq3iir32x32_df2_handle_t bq3iir32x32_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);
bq3iirf_df1_handle_t bq3iirf_df1_init(void * objmem, int M,
            const float32_t* coef_sos, int16_t gain );
bq3iirf_df2_handle_t bq3iirf_df2_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 32x16, or Q1.30 for 32x16 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 bq3iir16x16_df1(bq3iir16x16_df1_handle_t _bqriir,
                void *  s,int16_t *  r,const int16_t *x, int N);
void bq3iir16x16_df2(bq3iir16x16_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 bq3iir32x16_df1(bq3iir32x16_df1_handle_t _bqriir,
                void *  s,int32_t *  r,const int32_t *x, int N);
void bq3iir32x16_df2(bq3iir32x16_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 bq3iir32x32_df1(bq3iir32x32_df1_handle_t _bqriir,
                void *  s,int32_t *  r,const int32_t *x, int N);
void bq3iir32x32_df2(bq3iir32x32_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 _bqriir,
                  float32_t    * z, const float32_t * x, int N);
void bqriirf_df2t (bqriirf_df2t_handle_t _bqriir,
                  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);
void bq3iirf_df1 (bq3iirf_df1_handle_t,
                  float32_t    * r, const float32_t * x, int N);
void bq3iirf_df2 (bq3iirf_df2_handle_t,
                  float32_t    * r, const float32_t * x, int N);
```

Function	Scratch memory, bytes
bqriir16x16_df1	BQRIIR16X16_DF1_SCRATCH_SIZE(M)
bqriir16x16_df2	BQRIIR16X16_DF2_SCRATCH_SIZE(M)
bq3iir16x16_df1	BQ3IIR16X16_DF1_SCRATCH_SIZE(M)
bq3iir16x16_df2	BQ3IIR16X16_DF2_SCRATCH_SIZE(M)
bqriir24x24_df1	BQRIIR24X24_DF1_SCRATCH_SIZE(M)
bqriir24x24_df2	BQRIIR24X24_DF2_SCRATCH_SIZE(M)
bqriir32x16_df1	BQRIIR32X16_DF1_SCRATCH_SIZE(M)
bqriir32x16_df2	BQRIIR32X16_DF2_SCRATCH_SIZE(M)
bq3iir32x16_df1	BQ3IIR32X16_DF1_SCRATCH_SIZE(M)
bq3iir32x16_df2	BQ3IIR32X16_DF2_SCRATCH_SIZE(M)
bqriir32x32_df1	BQRIIR32X32_DF1_SCRATCH_SIZE(M)
bqriir32x32_df2	BQRIIR32X32_DF2_SCRATCH_SIZE(M)
bq3iir32x32_df1	BQ3IIR32X32_DF1_SCRATCH_SIZE(M)
bq3iir32x32_df2	BQ3IIR32X32_DF2_SCRATCH_SIZE(M)

Type	Name	Size	Description
Input			
int16_t, int32_t, float32_t, complex_float	x	N	input samples, Q31, Q15 or floating point. For 3-way functions (bq3iirxxx), N is a number of triplets, so array size should be 3*N.
int	N		length of input sample block. For 3-way functions (bq3iirxxx), N is the number of triplets
Output			
int16_t, int32_t, float32_t, complex_float	r	N	output data, Q31, Q15 or floating point. For 3-way functions (bq3iirxxx), N is the number of triplets, so array size should be 3*N.
Temporary			
void*	s		scratch memory area (for fixed-point functions only), Minimum number of bytes depends on selected filter structure and precision. see table above If a particular macro returns zero, then the corresponding IIR doesn't require a scratch area and parameter s may hold zero

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 is 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.

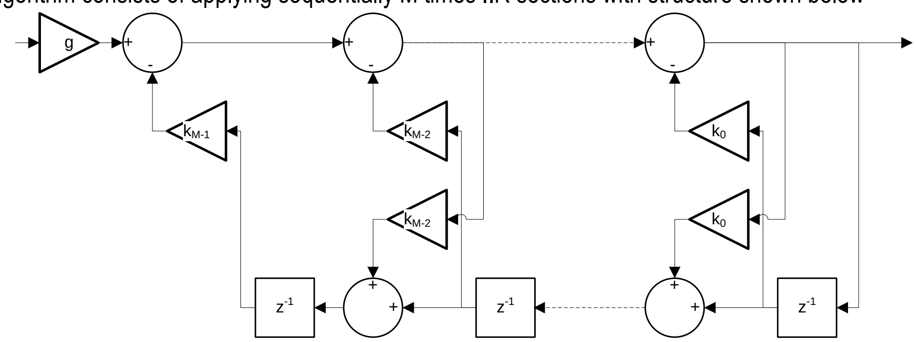
Precision

5 versions available:

Type	Description
16x16	16-bit data, 16-bit coefficients
24x24	24-bit data, 24-bit coefficients
32x16	32-bit data, 16-bit coefficients
32x32	32-bit data, 32-bit coefficients
f	floating point

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, Q15, Q31 or floating point
`int`	`N`		length of input sample block
Output			
`int16_t, f24, int32_t or float32_t`	`r`	`N`	output data, Q15, Q31 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 Vector Mathematics

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

`scl_log2f, scl_lognf, scl_log10f, scl_sinef, scl_cosinef, scl_tanf, scl_atanf, scl_atan2f, scl_antilog2f, scl_antilognf, scl_antilog10f, scl_asinf, scl_acosf, scl_sqrtf, scl_rsqrtf`

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 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

5 versions available:

Type	Description
16x16	16x16-bit data, 64-bit output for regular version and 32-bit for fast version
24x24	24x24-bit data, 64-bit output
32x16	32x16-bit data, 64-bit output
32x32	32x32-bit data, 64-bit output
f	floating point

Algorithm

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

Prototype

```
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_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_dot32x2_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			
`f24, int32_t, int16_t, float32_t`	`x`	`N`	input data, Q31, Q15 or floating point
`f24,int16_t, float32_t`	`y`	`N`	input data, 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_dot24x24, vec_dot32x16, vec_dot32x32, vec_dot16x16, vec_dotf`): None

Faster versions (`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.

2.3.2 Vector Addition

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 versions available:

Type	Description
32x32	32-bit inputs, 32-bit output
24x24	24-bit inputs, 24-bit output
16x16	16-bit inputs, 16-bit output
f	floating point

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

2.3.3 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 versions available:

Type	Description
24x24	24x24-bit data, 64-bit output
32x32	32x32-bit data, 64-bit output
16x16	16x16-bit data, 64-bit output
f	floating point

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: 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`)

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

2.3.4 Vector Scaling with Saturation

Description

These routines implement shift with saturation of data values in the vector, by the 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.
Function 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_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_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

4 versions available:

Type	Description
24x24	24-bit input, 32-bit output
32x32	32-bit input, 32-bit output
16x16	16-bit input, 16-bit output
f	floating point

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_t *  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_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_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_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_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

2.3.5 Reciprocal

Description

Fixed point 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.

For fixed point functions, 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

Floating point routines operate with standard floating-point numbers. Functions return +/-infinity on zero or denormalized input and provide accuracy of 1 ULP.

Precision

4 versions available:

Type	Description
32x32	32-bit input, 32-bit output.
24x24	24-bit input, 24-bit output.
16x16	16-bit input, 16-bit output.
f	floating point

Algorithm

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

$y_n = 1/x_n, n = \overline{0 \ldots N-1}$ for floating point functions

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)
void vec_recipf     (float32_t* y   ,               const float32_t* x, 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`	`N`		length of vectors
Output			
`f24, int32_t or int16_t`	`frac`	`N`	fractional part of result, Q(31-`exp`) or Q(15-`exp`) (fixed point functions)
`int16_t`	`exp`	`N`	exponent of result (fixed point functions)
`float32_t`	`y`	`N`	result (floating point function)

Returned value

None

Restrictions

`x,frac,exp` should not overlap

Scalar versions

Prototype

```
uint32_t  scl_recip32x32 (int32_t x)
uint32_t  scl_recip24x24 (f24 x)
uint32_t  scl_recip16x16 (int16_t x)
float32_t scl_recipf     (float32_t x)
```

Arguments

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

Returned value

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

2.3.6 Division

Description

Fixed point 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

For fixed point functions, 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

Floating point routines operate with standard floating-point numbers. Functions return +/-infinity in case of overflow and provide accuracy of 2 ULP.

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

Precision

4 versions available:

Type	Description
32x32	32-bit inputs, 32-bit output.
24x24	24-bit inputs, 24-bit output.
16x16	16-bit inputs, 16-bit output.
f	floating point

Algorithm

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

$z_n = x_n / y_n, n = \overline{0...N-1}$ for floating point functions

Prototype

```
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_dividef
                (float32_t * z,
                 const float32_t *  x, const float32_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			
`f24, int32_t, int16_t, float32_t`	`x`	`N`	nominator,Q31, Q15, floating point

`f24, int32_t, int16_t, float32_t`	`y`	`N`	denominator,Q31, Q15, floating point
`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`) (for fixed point functions)
`int16_t`	`exp`	`N`	exponents of result (for fixed point functions)
`float32_t`	`z`	`N`	result (for floating point function)

Returned value

none

Restrictions

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

For faster versions (`vec_divide32x3_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

```
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)
float32_t scl_dividef     (float32_t x, float32_t y)
```

Arguments

Type	Name	Description
Input		
`int32_t, f24, int16_t, float32_t`	`x`	nominator, Q31, Q15, floating point
`int32_t, f24, int16_t, float32_t`	`y`	denominator, Q31, Q15, floating point

Returned value

packed value (for fixed point functions):
`scl_divide24x24(),scl_divide32x32()`:
bits 23…0 fractional part
bits 31…24 exponent
`scl_divide16x16()`:
bits 15…0 fractional part
bits 31…16 exponent

2.3.7 Logarithm

Description

Different kinds of logarithm (base 2, natural, base 10). 32 and 24-bit fixed point functions interpret input as Q16.15 and represent results in Q25 format or return 0x80000000 on negative of zero input. 16-bit fixed-point functions interpret input as Q8.7 and represent result in Q3.12 or return 0x8000 on negative of zero input

Accuracy :

16x16 functions	`2 LSB`
`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

4 versions available:

Type	Description
16x16	16-bit inputs, 16-bit outputs
24x24	24-bit inputs, 24-bit outputs
32x32	32-bit inputs, 32-bit outputs
f	floating point

Algorithm

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

Prototypes

```
void vec_log2_16x16 (      int16_t *  y, const int16_t *  x, int N);
void vec_logn_16x16 (      int16_t *  y, const int16_t *  x, int N);
void vec_log10_16x16(      int16_t *  y, const int16_t *  x, int N);
void vec_log2_24x24 (      f24 *  y, const f24 *  x, int N);
void vec_logn_24x24 (      f24 *  y, const f24 *  x, int N);
void vec_log10_24x24(      f24 *  y, const f24 *  x, int N);
void vec_log2_32x32 (      int32_t *  y, const int32_t *  x, int N);
void vec_logn_32x32 (      int32_t *  y, const int32_t *  x, int N);
void vec_log10_32x32(      int32_t *  y, const int32_t *  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			
`int16_t, f24, int32_t, float32_t`	`x`	`N`	input data, Q16.15 (32 or 24-bit functions), Q8.7 (16-bit functions) or floating point
`int`	`N`		length of vectors
Output			

int16_t, f24, int32_t, float32_t	y	N	Q6.25 (32 or 24-bit functions), Q3.12 (16-bit functions) or floating point

Returned value none

Restrictions x,y – should not overlap

Scalar versions

Prototypes

```
int16_t scl_log2_16x16 (int16_t x);
int16_t scl_logn_16x16 (int16_t x);
int16_t scl_log10_16x16(int16_t x);
f24 scl_log2_24x24 (f24 x);
f24 scl_logn_24x24 (f24 x);
f24 scl_log10_24x24(f24 x);
int32_t scl_log2_32x32 (int32_t x);
int32_t scl_logn_32x32 (int32_t x);
int32_t scl_log10_32x32(int32_t 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		
int16_t, f24, int32_t, float32_t	x	input data, Q16.15 (32 or 24-bit functions), Q8.7 (16-bit functions) or floating point

Returned value result, Q6.25 (32 or 24-bit functions), Q3.12 (16-bit functions) or floating point

2.3.8 Antilogarithm

Description

These routines calculate antilogarithm (base2, natural and base10). 32 and 24-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. 16-bit fixed-point functions accept inputs in Q3.12 and form outputs in Q8.7 format and return 0x7FFF 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

4 versions available:

Type	Description
16x16	16-bit inputs, 16-bit outputs. Accuracy: 2 LSB
24x24	24-bit inputs, 24-bit outputs. Accuracy: 8e-6*y+1LSB
32x32	32-bit inputs, 32-bit outputs. Accuracy: 8e-6*y+1LSB
f	floating point. Accuracy: 2 ULP

Algorithm

$$y_n = 2^{x_n}$$

$$y_n = e^{x_n}$$

$$y_n = 10^{x_n}$$

Prototype

```
void vec_antilog2_16x16 (int16_t *  y, const int16_t *  x, int N);
void vec_antilogn_16x16 (int16_t *  y, const int16_t *  x, int N);
void vec_antilog10_16x16(int16_t *  y, const int16_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_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_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			
`int16_t, f24,int32_t, float32_t`	`x`	`N`	input data, Q6.25 (for 32 and 24-bit functions), Q3.12 (for 16-bit functions) or floating point
`int`	`N`		length of vectors
Output			
`int16_t, f24,int32_t, float32_t`	`y`	`N`	output data, Q16.15 (for 32 and 24-bit functions), Q8.7 (for 16-bit functions) 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

```
int16_t scl_antilog2_16x16 (int16_t x);
int16_t scl_antilogn_16x16 (int16_t x);
int16_t scl_antilog10_16x16(int16_t x);
f24 scl_antilog2_24x24 (f24 x);
f24 scl_antilogn_24x24 (f24 x);
```

```
f24 scl_antilog10_24x24(f24 x);
int32_t scl_antilog2_32x32 (int32_t x);
int32_t scl_antilogn_32x32 (int32_t x);
int32_t scl_antilog10_32x32(int32_t 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		
`int16_t`, `f24`, `int32_t`, `float32_t`	`x`	input data, Q6.25 (for 32 and 24-bit functions), Q3.12 (for 16-bit functions) or floating point

Returned value

result, Q16.15 (for 32 and 24-bit functions), Q8.7 (for 16-bit functions) or floating point

2.3.9 Square Root

Description

These routines calculate square root.
NOTES:

1. Fixed point functions return 0x80000000 (for 24 and 32-bit functions), 0x8000 (for 16-bit functions) on negative argument
2. For floating point function, whenever an input value is negative, functions raise the "invalid" floating-point exception, assign EDOM to errno and set output value to NaN. Negative zero is considered as a valid input, the result is also -0

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

Precision

4 versions 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)
32x32	32-bit inputs, 32-bit output. Accuracy: (2.6e-7*y+1LSB)
f	floating point. Accuracy 1 ULP

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_sqrt24x24_fast(  f24 *      y, const f24      *  x, int N);
void vec_sqrt32x32_fast(  int32_t*   y, const int32_t  *  x, int N);
void vec_sqrtf     (      float32_t* y, const float32_t*  x, int N);
```

Arguments

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

Returned value

none

Restrictions

Regular versions (`vec_sqrt16x16, vec_sqrt24x24, vec_sqrt32x32`):
`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);
float32_t scl_sqrtf    (float32_t x);
```

Arguments

Type	Name	Description
Input		
`f24 or int32_t, float32_t`	`x`	input data, Q31, Q15 or floating point

Returned value

result, Q31, Q15 or floating point

2.3.10 Reciprocal Square Root

Description

These routines compute reciprocals of positive square root.
Whenever an input value is negative, functions raise the "invalid" floating-point exception, assign `EDOM` to `errno` and set output value to `NaN`. For `x[n]==+/-0`, functions set output to +/-infinity, raise the "divide by zero" floating-point exception, and assign `ERANGE` to `errno`.

Precision

1 version available:

Type	Description
f	floating point. Accuracy 2 ULP

Algorithm

$$y_n = 1/\sqrt{x_n}$$

Prototype

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

Arguments

Type	Name	Size	Description
Input			
`float32_t`	x	N	input data
`int`	N		length of vectors
Output			
`float32_t`	y	N	output data

Returned value

none

Restrictions

`x,y` – should not overlap

Scalar versions

Prototypes

```
float32_t scl_rsqrtf    (float32_t x);
```

Arguments

Type	Name	Description
Input		
`float32_t`	x	input data

2.3.11 Sine/Cosine

Description

Fixed-point functions calculate `sin(pi*x)` or `cos(pi*x)` for numbers written in Q31 or Q15 format. Return results in the same format. Floating point functions compute `sin(x)` or `cos(x)`.
Two versions of functions available: regular version (`vec_sine16x16, vec_cosine16x16, 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`. I

Precision

4 versions available:

Type	Description
16x16	16-bit inputs, 16-bit output. Accuracy: 2 LSB
24x24	24-bit inputs, 24-bit output. Accuracy: 74000(3.4e-5)
32x32	32-bit inputs, 32-bit output. Accuracy: 1700 (7.9e-7)
f	floating point. Accuracy 2 ULP

Algorithm

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

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

Prototypes

```
void vec_sine16x16   (int16_t  *  y, const int16_t  * x, int N);
void vec_cosine16x16 (int16_t  *  y, const int16_t  * x, int N);
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			
`int16_t, f24, int32_t, float32_t`	`x`	`N`	input data, Q15, Q31 or floating point
`int`	`N`		length of vectors
Output			
`int16_t, f24, int32_t, float32_t`	`y`	`N`	Result, Q15, Q31 or floating point

Returned value

None

Restrictions

Regular versions (`vec_sine16x16, vec_cosine16x16, 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

```
int16_t scl_sine16x16   (int16_t x);
int16_t scl_cosine16x16 (int16_t x);
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		
`int16_t, f24, int32_t, float32_t`	`x`	input data, Q15, Q31 or floating point

Returned value

result, Q15, Q31 or floating point

2.3.12 Tangent

Description

Fixed point functions calculate `tan(pi*x)` for number written in Q15 or 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

4 versions available:

Type	Description
16x16	16-bit inputs (Q15), 16-bit outputs (Q8.7). Accuracy: 1 LSB
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`
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

Algorithm

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

Prototype

```
void vec_tan16x16 (int16_t*    y, const int16_t  * x, int N);
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			
`int16_t, f24, int32_t, float32_t`	`x`	`N`	input data, Q15, Q31 or floating point
`int`	`N`		length of vectors
Output			
`int16_t, int32_t, float32_t`	`y`	`N`	result, Q8.7 (16-bit function), Q16.15 (24 or 32-bit functions) 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

```
int16_t scl_tan16x16 (int16_t x);
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		
`int16_t, f24, int32_t, float32_t`	`x`	input data, Q15, Q31 or floating point

Returned value	result, Q8.7 (16-bit function), Q16.15 (24 or 32-bit functions) or floating point

2.3.13 Arc Sine/Cosine

Description

The arc sine/cosine functions return the arc sine/cosine of x. Output is in radians. For floating-point routines, input value should belong to [-1,1], otherwise the functions raise the “invalid" floating-point exception, assign `EDOM` to `errno` and return `NaN`..

Precision

1 version available:

Type	Description
f	floating point. Accuracy: 2 ULP

Algorithm

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

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

Prototype

```
void vec_asinf     (float32_t *  z, const float32_t *  x, int N );
void vec_acosf     (float32_t *  z, const float32_t *  x, int N );
```

Arguments

Type	Name	Size	Description
Input			
`float32_t`	`x`	`N`	input data
`int`	`N`		length of vectors
Output			
`float32_t`	`z`	`N`	result

Returned value

None

Restrictions

`x,z` should not overlap

Scalar versions

Prototype

```
float32_t scl_asinf (float32_t x);
float32_t scl_acosf (float32_t x);
```

Arguments

Type	Name	Description
Input		
`float32_t`	`x`	input data

Returned value

result

2.3.14 Arctangent

Description

Functions calculate arctangent of number. Fixed point functions scale output by `pi` which corresponds to the real phases `+pi/4` and represent input and output in Q15 or Q31

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

Precision

4 versions available:

Type	Description
16x16	16-bit inputs, 16-bit output. Accuracy: 2 LSB
24x24	24-bit inputs, 24-bit output. Accuracy: 74000 (3.4e-5)
32x32	32-bit inputs, 32-bit output. Accuracy: 42 (2.0e-8)
f	floating point. Accuracy: 2 ULP

Algorithm

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

Prototype

```
void vec_atan16x16 (int16_t   *  z, const int16_t   *  x, int N );
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			
`int16_t, f24, int32_t, float32_t`	`x`	`N`	input data, Q15, Q31 or floating point
`int`	`N`		length of vectors
Output			
`int16_t, f24, int32_t, float32_t`	`z`	`N`	result, Q15, 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

```
int16_t scl_atan16x16 (int16_t x);
f24 scl_atan24x24 (f24 x);
int32_t scl_atan32x32 (int32_t x);
float32_t scl_atanf (float32_t x);
```

Arguments

Type	Name	Description
Input		
`int16_t, f24, int32_t, float32_t`	`x`	input data, Q15, Q31 or floating point

Returned value

result, Q15, Q31 or floating point

2.3.15 Full Quadrant Arctangent

Description

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

NOTE:

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

Special cases for floating point

y	x	result	extra conditions
+/-0	-0	+/-pi	
+/-0	+0	+/-0	
+/-0	x	+/-pi	x<0
+/-0	x	+/-0	x>0
y	+/-0	-pi/2	y<0
y	+/-0	pi/2	y>0
+/-y	-inf	+/-pi	finite y>0
+/-y	+inf	+/-0	finite y>0
+/-inf	x	+/-pi/2	finite x
+/-inf	-inf	+/-3*pi/4	
+/-inf	+inf	+/-pi/4	

Precision

2 versions available:

Type	Description
16x16	16-bit input, 16-bit output. Accuracy: 2 LSB
f	floating point. Accuracy: 2 ULP

Algorithm

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

Prototype

```
void vec_atan2_16x16 (int16_t * z, const int16_t * y, const int16_t * x,int N);
void vec_atan2f (float32_t * z, const float32_t * y, const float32_t * x,int N);
```

Arguments

Type	Name	Size	Description
Input			
`int16_t, float32_t`	`x`	`N`	input data, Q15 or floating point
`int16_t, float32_t`	`y`	`N`	input data, Q15 or floating point
`int`	`N`		length of vectors
Output			
`int16_t, float32_t`	`z`	`N`	result, Q15 or floating point

Returned value

None

Restrictions

`x, y, z` should not overlap

Scalar versions

Prototype

```
int16_t   scl_atan2_16x16(int16_t y, int16_t x);
float32_t scl_atan2f (float32_t y, float32_t x);
```

Arguments

Type	Name	Description
Input		
`int16_t, float32_t`	`x`	input data, Q15 or floating point

int16_t, float32_t	y	input data, Q15 or floating point

Returned value result, Q15 or floating point

2.3.16 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 the 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 bit exact 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 versions available:

Type	Description
32	32-bit inputs
24	24-bit inputs
16	16-bit inputs
f	floating point inputs

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

2.3.17 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 versions available:

Type	Description
32x32	32-bit data, 32-bit output
24x24	24-bit data, 24-bit output
16x16	16-bit data, 16-bit output
f	floating point

Algorithm

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

or

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

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

2.3.18 Polynomial approximation

Description

Functions calculate polynomial approximation for all values from given vector. Fixed point functions take polynomial coefficients in Q15 or 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 than 1. To avoid this negative effect, 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

4 versions available:

Type	Description
16x16	16-bit inputs, 16-bit coefficients, 16-bit output.
24x24	24-bit inputs, 24-bit coefficients, 24-bit output.
32x32	32-bit inputs, 32-bit coefficients, 32-bit output.
f	floating point

Algorithm

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

Prototype

```
void vec_poly4_16x16 (int16_t *  z, const int16_t *  x,
                      const int16_t *  c, int lsh, int N );
void vec_poly8_16x16 (int16_t *  z, const int16_t *  x,
                      const int16_t *  c, int lsh, int N );
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_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 );
void vec_poly8f      (float32_t *  z, const float32_t *  x,
                      const float32_t *  c, int N );
```

Arguments

Type	Name	Size	Description
Input			
`int16_t, f24, int32_t, float32_t`	`x`	`N`	input data, Q15, Q31 or floating point
`int16_t, f24 or int32_t`	`c`	`5 or 9`	coefficients (5 coefficients for `vec_poly4_xxx` and 9 coefficients for `vec_poly8_xxx`), Q15, Q31 or floating point
`int`	`lsh`		additional left shift for result
`int`	`N`		length of vectors
Output			
`int16_t, f24, int32_t, float32_t`	`z`	`N`	result, Q15, 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
```

2.3.19 Integer to Float Conversion

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

Precision 1 version available:

Type	Description
f	32-bit input, floating point output

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
`int`	`t`		scale factor
`int`	`N`		length of vectors
Output			
`float32_t`	`y`	`N`	scaled floating point values

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

Returned value result, floating point

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

2.3.20 Float to Integer Conversion

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

Precision
1 version available:

Type	Description
f	floating point input, 32-bit output

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`	scaled floating point values

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

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

2.3.21 Rounding

Description

Routines make floating point round to integral value. Input data are rounded up/down to the nearest integral value but maintained in the same floating-point format.

Precision

1 version available:

Type	Description
f	floating point input/output

Algorithm

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

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

Prototype

```
void   vec_float2floor
          (  float32_t *  y,
             const float32_t *  x,
             int N);
void   vec_float2ceil
          (  float32_t *  y,
             const float32_t *  x,
             int N);
```

Arguments

Type	Name	Size	Description
Input			
`float32_t`	`x`	`N`	input data, floating point
`int`	`N`		length of vectors
Output			
`float32_t`	`y`	`N`	rounded floating point values

Returned value

None

Restrictions

`x,y` should not overlap

Scalar version

Prototype

```
float32_t scl_float2floor  (float32_t x);
float32_t scl_float2ceil   (float32_t x);
```

Arguments

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

Returned value

result

Restrictions

none

2.3.22 Complex magnitude

Description

Routines compute complex magnitude or its reciprocal

Precision

1 version available:

Type	Description
f	floating point input, 32-bit output

Algorithm

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

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

Prototype

```
void        vec_complex2mag
           (float32_t  *  y,
            const complex_float  *  x,
            int N);
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	magnitude or its reciprocal

Returned value

None

Restrictions

None

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 Matrix Operations

2.4.1 Matrix Multiply

Description

These functions compute the expression `z = 2^lsh * x * y` for the matrices `x` and `y`. The columnar dimension of `x` must match the row dimension of `y`. The resulting matrix has the same number of rows as `x` and the same number of columns as `y`.
NOTES:
In the fixed-point routines, rows of matrices `z` and `y` may be stored in non consecutive manner. Matrix `x` will have all the elements in contiguous memory locations.

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_MPY32X32(M,N,P)`, `SCRATCH_MTX_MPY24X24(M,N,P)`, `SCRATCH_MTX_MPY16X16(M,N,P)`

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

Precision

4 versions available:

Type	Description
32x32	32-bit inputs, 32-bit output
24x24	24-bit inputs, 24-bit output
16x16	16-bit inputs, 16-bit output
f	floating point

Algorithm

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

Prototype

```
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_mpy24x24 (  void* pScr,
                     f24**  z,
               const f24*   x,
               const f24**  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_mpy32x32_fast (  int32_t**  z,
               const int32_t *   x,
               const int32_t **  y,
               int M, int N, int P, int lsh );
void mtx_mpy24x24_fast (  f24**  z,
               const f24*   x,
               const f24**  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			
`int32_t, f24, int16_t, float32_t`	`x`	`M*N`	input matrix,Q31 or Q15
`int32_t, f24, int16_t, float32_t`	`y`	`N*P`	input matrix `y`. For fixed point routines, these are `N` vectors of size `P`, Q31 or Q15. For floating point, this is just a matrix of size `NxP`.
`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			
`int32_t, f24, int16_t, float32_t`	`z`	`M*P`	output matrix `z`. For fixed point routines, these are `M` vectors of size `P` Q31 or Q15. For floating point, this is single matrix of size `MxP`
Temporary			
`void*`	`pScr`		Scratch memory area with size in bytes defined by macros `SCRATCH_MTX_MPY32X32`, `SCRATCH_MTX_MPY24X24`, `SCRATCH_MTX_MPY16X16`

Returned value

none

Restrictions

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

For faster routines (`mtx_mpy32x32_fast, mtx_mpy24x24_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`

2.4.2 Matrix by Vector Multiply

Description

These functions compute the expression `z = 2^lsh * x * y` for the matrices `x` and vector `y`.

Two versions of functions available: regular version (`mtx_vecmpy32x32, mtx_vecmpy24x24, mtx_vecmpy16x16, mtx_vecmpyf`) with arbitrary arguments and faster version (`mtx_vecmpy32x32_fast, mtx_vecmpy24x24_fast, mtx_vecmpy16x16_fast, mtx_vecmpyf_fast`) that apply some restrictions.

Precision

4 versions available:

Type	Description
32x32	32-bit inputs, 32-bit output
24x24	24-bit inputs, 24-bit output
16x16	16-bit inputs, 16-bit output
f	floating point

Algorithm

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

Prototype

```
void mtx_vecmpy32x32 (  int32_t*  z,
               const int32_t *  x,
               const int32_t *  y,
               int M, int N, int lsh);
void mtx_vecmpy24x24 (  f24*  z,
               const f24*  x,
               const f24*  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_vecmpy32x32_fast (  int32_t*  z,
               const int32_t *  x,
               const int32_t *  y,
               int M, int N, int lsh);
void mtx_vecmpy24x24_fast (  f24*  z,
               const f24*  x,
               const f24*  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			
`int32_t, f24, int16_t, float32_t`	`x`	`M*N`	input matrix, Q31, Q15 or floating point
`int32_t, f24, 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			
`int32_t, f24, int16_t, float32_t`	`z`	`M`	output vector, Q31, Q15 or floating point

Returned value

None

Restrictions

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

For faster routines (`mtx_vecmpy32x32_fast, mtx_vecmpy24x24_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`

2.4.3 Operations with Small Matrices

Description

These functions implement basic operations under the sequence of small square matrices. Fixed point data are interpreted as Q15 or Q31 and results might be saturated.
NOTE:
Determinant is computed recursively via minors of submatrices. So, in the fixed-point routines, intermediate results might be saturated although final result is in range. To avoid this saturation, right shift might be applied at the first stage of computations. It means that final result would be represented in Q(15-rsh) or Q(31-rsh) respectively. Ad-hoc formula for rsh is rsh>=N-2 for real matrices and rsh>=N-1 for complex matrices.

Precision

3 versions available:

Type	Description
16x16	16-bit input, 16-bit output (real and complex)
32x32	32-bit input, 32-bit output (real and complex)
f	floating point (real and complex)

Algorithm

$$z_l = x_l + y_l$$

$$z_l = x_l - y_l$$

$$z_l = x_l \cdot y_l$$

$$z_l = x_l^T$$

$$d_l = \det(x_l) \cdot 2^{-rsh}$$

$l = \overline{0...L-1}$, x_l, y_l, z_l - matrices of size `N`x`N`

Prototypes (addition, subtraction, multiply)

Real matrices:

```
void fun(int16_t   *z, const int16_t   *x, const int16_t   *y, int L);
void fun(int32_t   *z, const int32_t   *x, const int32_t   *y, int L);
void fun(float32_t *z, const float32_t *x, const float32_t *y, int L);
```

Complex matrices:

```
void fun(complex_fract16   *z,
         const complex_fract16 *x, const complex_fract16 *y, int L);
void fun(complex_fract32   *z,
         const complex_fract32 *x, const complex_fract32 *y, int L);
void fun(complex_float     *z,
         const complex_float   *x, const complex_float   *y, int L);
```

Data type	Function name		
	N=2	N=3	N=4
Matrix addition			
int16_t	mtx_add2x2_16x16	mtx_add3x3_16x16	mtx_add4x4_16x16
int32_t	mtx_add2x2_32x32	mtx_add3x3_32x32	mtx_add4x4_32x32
float32_t	mtx_add2x2f	mtx_add3x3f	mtx_add4x4f
complex_fract16	cmtx_add2x2_16x16	cmtx_add3x3_16x16	cmtx_add4x4_16x16
complex_fract32	cmtx_add2x2_32x32	cmtx_add3x3_32x32	cmtx_add4x4_32x32
complex_float	cmtx_add2x2f	cmtx_add3x3f	cmtx_add4x4f
Matrix subtraction			
int16_t	mtx_sub2x2_16x16	mtx_sub3x3_16x16	mtx_sub4x4_16x16
int32_t	mtx_sub2x2_32x32	mtx_sub3x3_32x32	mtx_sub4x4_32x32
float32_t	mtx_sub2x2f	mtx_sub3x3f	mtx_sub4x4f
complex_fract16	cmtx_sub2x2_16x16	cmtx_sub3x3_16x16	cmtx_sub4x4_16x16
complex_fract32	cmtx_sub2x2_32x32	cmtx_sub3x3_32x32	cmtx_sub4x4_32x32
complex_float	cmtx_sub2x2f	cmtx_sub3x3f	cmtx_sub4x4f

Prototypes (multiply)

Real matrices:

```
void fun(int16_t   *z, const int16_t   *x, const int16_t   *y, int rsh, int L);
void fun(int32_t   *z, const int32_t   *x, const int32_t   *y, int rsh, int L);
void fun(float32_t *z, const float32_t *x, const float32_t *y,          int L);
```

Complex matrices:

```
void fun(complex_fract16   *z,
         const complex_fract16 *x, const complex_fract16 *y, int rsh, int L);
void fun(complex_fract32   *z,
         const complex_fract32 *x, const complex_fract32 *y, int rsh, int L);
void fun(complex_float     *z,
         const complex_float   *x, const complex_float   *y, int L);
```

Data type	Function name		
	N=2	N=3	N=4
Matrix multiply			
int16_t	mtx_mul2x2_16x16	mtx_mul3x3_16x16	mtx_mul4x4_16x16
int32_t	mtx_mul2x2_32x32	mtx_mul3x3_32x32	mtx_mul4x4_32x32
float32_t	mtx_mul2x2f	mtx_mul3x3f	mtx_mul4x4f
complex_fract16	cmtx_mul2x2_16x16	cmtx_mul3x3_16x16	cmtx_mul4x4_16x16
complex_fract32	cmtx_mul2x2_32x32	cmtx_mul3x3_32x32	cmtx_mul4x4_32x32
complex_float	cmtx_mul2x2f	cmtx_mul3x3f	cmtx_mul4x4f

Prototypes (transpose)

Real matrices:

```
void fun(int16_t   *z, const int16_t   *x, int L);
void fun(int32_t   *z, const int32_t   *x, int L);
void fun(float32_t *z, const float32_t *x, int L);
```

Complex matrices:

```
void fun(complex_fract16   *z, const complex_fract16 *x, int L);
void fun(complex_fract32   *z, const complex_fract32 *x, int L);
void fun(complex_float     *z, const complex_float   *x, int L);
```

Data type	Function name		
	N=2	N=3	N=4
int16_t	mtx_tran2x2_16x16	mtx_tran3x3_16x16	mtx_tran4x4_16x16
int32_t	mtx_tran2x2_32x32	mtx_tran3x3_32x32	mtx_tran4x4_32x32
float32_t	mtx_tran2x2f	mtx_tran3x3f	mtx_tran4x4f
complex_fract16	cmtx_tran2x2_16x16	cmtx_tran3x3_16x16	cmtx_tran4x4_16x16
complex_fract32	cmtx_tran2x2_32x32	cmtx_tran3x3_32x32	cmtx_tran4x4_32x32
complex_float	cmtx_tran2x2f	cmtx_tran3x3f	cmtx_tran4x4f

Prototypes (determinant)

Real matrices:

```
void fun(int16_t   *d, const int16_t   *x, int rsh, int L);
void fun(int32_t   *d, const int32_t   *x, int rsh, int L);
void fun(float32_t *d, const float32_t *x,          int L);
```

Complex matrices:

```
void fun(complex_fract16   *d, const complex_fract16 *x, int rsh, int L);
void fun(complex_fract32   *d, const complex_fract32 *x, int rsh, int L);
void fun(complex_float     *d, const complex_float   *x,          int L);
```

Data type	Function name		
	N=2	N=3	N=4
int16_t	mtx_det2x2_16x16	mtx_det3x3_16x16	mtx_det4x4_16x16
int32_t	mtx_det2x2_32x32	mtx_det3x3_32x32	mtx_det4x4_32x32
float32_t	mtx_det2x2f	mtx_det3x3f	mtx_det4x4f
complex_fract16	cmtx_det2x2_16x16	cmtx_det3x3_16x16	cmtx_det4x4_16x16
complex_fract32	cmtx_det2x2_32x32	cmtx_det3x3_32x32	cmtx_det4x4_32x32
complex_float	cmtx_det2x2f	cmtx_det3x3f	cmtx_det4x4f

Arguments

Type	Name	Size	Description
Input			
`int16_t`, `int32_t`, `float32_t`, `complex_fract16`, `complex_fract32`, `complex_float`	`x`	`[L][N*N]`	`L` input matrices
`int16_t`, `int32_t`, `float32_t`, `complex_fract16`, `complex_fract32`, `complex_float`	`y`	`[L][N*N]`	`L` input matrices (for addition, subtraction, multiply functions)
`int`	`rsh`		right shift for fixed-point multiply and determinant functions
`int`	`L`		number of matrices
Output			
`int16_t`, `int32_t`, `float32_t`, `complex_fract16`, `complex_fract32`, `complex_float`	`z`	`[L][N*N]`	`L` output matrices (for addition, subtraction, multiply, transpose functions)
`int16_t`, `int32_t`, `float32_t`, `complex_fract16`, `complex_fract32`, `complex_float`	`d`	`L`	determinants for `L` matrices (for determinant functions)

Returned value

none

Restrictions

`rsh` should be in range 0..15
`x,y,z` should not overlap

2.4.4 Matrix Inverse

Description

These functions implement in-place matrix inversion by Gauss elimination with full pivoting.

Precision

1 version available:

Type	Description
f	floating point (real and complex)

Algorithm

$y = x^{-1}$

Prototype

```
void mtx_inv2x2f(float32_t *x);
void mtx_inv3x3f(float32_t *x);
void mtx_inv4x4f(float32_t *x);
void cmtx_inv2x2f(complex_float *x);
void cmtx_inv3x3f(complex_float *x);
void cmtx_inv4x4f(complex_float *x);
```

Matrix dimension, N	Function
2	`mxt_inv2x2f, cmxt_inv2x2f`
3	`mxt_inv3x3f, cmxt_inv3x3f`
4	`mxt_inv4x4f, cmxt_inv4x4f`

Arguments

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

Returned value none

Restrictions none

2.4.5 Quaternion to Rotation Matrix Conversion

Description These functions convert sequence of unit quaternions to corresponding rotation matrices,

Precision 3 versions available:

Type	Description
16x16	16-bit input, 16-bit output
32x32	32-bit input, 32-bit output
f	floating point

Algorithm

$$R_l = \begin{bmatrix} q_0^2+q_1^2-q_2^2-q_3^2 & 2q_1q_2+2q_0q_3 & 2q_1q_3-2q_0q_2 \\ 2q_1q_2-2q_0q_3 & q_0^2-q_1^2+q_2^2-q_3^2 & 2q_2q_3+2q_0q_1 \\ 2q_1q_3+2q_0q_2 & 2q_2q_3-2q_0q_1 & q_0^2-q_1^2-q_2^2+q_3^2 \end{bmatrix}, l=\overline{0 \ldots L-1}$$

where

$q_{0\ldots3}$ - elements of l-th quaternion

Prototype

```
void q2rot_16x16(int16_t   *r, const int16_t   *q, int L);
void q2rot_32x32(int32_t   *r, const int32_t   *q, int L);
void q2rotf     (float32_t *r, const float32_t *q, int L);
```

Arguments

Type	Name	Size	Description
Input			
`int16_t`, `int32_t` `float32_t`	`q`	`[L][4]`	`L` quaternions
`int`	`L`		number of matrices
Output			
`int16_t`, `int32_t` `float32_t`	`r`	`[L][3*3]`	`L` rotation matrices

Returned value none

Restrictions `q,r` should not overlap

2.5 Fast Fourier Transforms

FFT functions make floating point, 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 radix-4/radix-2 in-place transformations so **input data may be damaged**.

Different types of data scaling options 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. 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 compared to static scaling;
- **static scaling (**`scalingOption = 3`**),** has better performance but lesser accuracy than dynamic scaling.

With dynamic scaling (`scalingOption = 2`), the input data are normalized in the first phase of the FFT, so that there is no overflow. In subsequent phases, the data is 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).

HiFi 1 provides specialized instructions to facilitate dynamic scaling in FFT. The AE_SAR register contains guard-bits information and shift-amount information, which is used for implementation of dynamic scaling. The AE_S16X4RNG* instructions scan the 16-bit data stream to detect the guard-bits available and preserve the information in the AE_SAR register in addition to store operation. AE_CALCRNG3 is the helper instruction to translate the guard-bits information to meaningful shift values and updates the shift-amount information in AE_SAR register. The intrinsics AE_ADDANDSUBRNG16RAS_S1 and AE_ADDANDSUBRNG16RAS_S2 use the shift-amount information in AE_SAR for fused addition and subtraction with arithmetic-right shift operation.
The instruction AE_CALCRNG3 is meant to be used after a series of AE_S16X4RNG* instructions, and before AE_ADDANDSUBRNG16RAS_S* intrinsics.

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
- scaling method selection is fixed at a single option
- twiddle factor tables are provided by user. A single table may be shared between FFTs/IFFTs of varying size
- 24-bit packed format is used for input/output/temporary data storage where 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 `vec_shift()/vec_shift32()` helps 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:
$t_{FFT} + t_{IFFT} + \log_2(N) \equiv t_{FFT} + t_{IFFT} + 31 - \mathrm{scl\_bexp32}(N)$

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	FFT/IFFT on complex data	0
0	FFT/IFFT on real data	0
1,2	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)`

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	Input signal < 2^23/(2*N), N - FFT size None None

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

Precision	Scaling options	Restrictions on the dynamic range of the input signal
	scaling later 3 – fixed scaling before each stage	 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
`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
`real24x24_ie_24p`	1 – 24-bit scaling	None
`cplxf_ie`		
`realf_ie`		
DCT		

`dct_16x16,`
`dct_24x24,`
`dct_32x16,`
`dct_32x32,`
	3 – fixed scaling before each stage	None

`dctf`		

2.5.1 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.

Precision

4 versions available:

Type	Description
32x32	32-bit input/outputs, 32-bit twiddles
24x24	24-bit input/outputs, 24-bit twiddles
32x16	32-bit input/outputs, 16-bit twiddles
16x16	16-bit input/outputs, 16-bit twiddles

Algorithm

$$y = FFT(x)$$

Prototype

```
int fft_cplx32x32(int32_t* y, int32_t * x, fft_handle_t h, int scalingOption)
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_cplx16x16(int16_t* y, int16_t * x, fft_handle_t h, int scalingOption)
```

FFT handles:

N	32x32	24x24	32x16	16x16
16	cfft32_16	cfft24_16	cfft16_16	cfft16x16_16
32	cfft32_32	cfft24_32	cfft16_32	cfft16x16_32
64	cfft32_64	cfft24_64	cfft16_64	cfft16x16_64
128	cfft32_128	cfft24_128	cfft16_128	cfft16x16_128
256	cfft32_256	cfft24_256	cfft16_256	cfft16x16_256
512	cfft32_512	cfft24_512	cfft16_512	cfft16x16_512
1024	cfft32_1024	cfft24_1024	cfft16_1024	cfft16x16_1024
2048	cfft32_2048	cfft24_2048	cfft16_2048	cfft16x16_2048
4096	cfft32_4096	cfft24_4096	cfft16_4096	cfft16x16_4096

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 2.5)
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 8-byte boundary

2.5.2 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.

Precision

4 versions available:

Type	Description
32x32	32-bit input/outputs, 32-bit twiddles
24x24	24-bit input/outputs, 24-bit twiddles
32x16	32-bit input/outputs, 16-bit twiddles
16x16	16-bit input/outputs, 16-bit twiddles

Algorithm

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

Prototype

```
int fft_real32x32(int32_t* y, int32_t* x, fft_handle_t h, int scalingOpt)
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_real16x16(int16_t* y, int16_t* x, fft_handle_t h, int scalingOpt)
```

FFT handles:

N	32x32	24x24	32x16	16x16
32	`rfft32_32`	`rfft24_32`	`rfft16_32`	`rfft16x16_32`
64	`rfft32_64`	`rfft24_64`	`rfft16_64`	`rfft16x16_64`
128	`rfft32_128`	`rfft24_128`	`rfft16_128`	`rfft16x16_128`
256	`rfft32_256`	`rfft24_256`	`rfft16_256`	`rfft16x16_256`
512	`rfft32_512`	`rfft24_512`	`rfft16_512`	`rfft16x16_512`
1024	`rfft32_1024`	`rfft24_1024`	`rfft16_1024`	`rfft16x16_1024`
2048	`rfft32_2048`	`rfft24_2048`	`rfft16_2048`	`rfft16x16_2048`
4096	`rfft32_4096`	`rfft24_4096`	`rfft16_4096`	`rfft16x16_4096`
8192	`rfft32_8192`	`rfft24_8192`	`rfft16_8192`	`rfft16x16_8192`

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 2.5)
Output			
`int32_t, 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

2.5.3 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

Precision

4 versions available:

Type	Description
32x32	32-bit input/outputs, 32-bit twiddles
24x24	24-bit input/outputs, 24-bit twiddles
32x16	32-bit input/outputs, 16-bit twiddles
16x16	16-bit input/outputs, 16-bit twiddles

Algorithm

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

Prototype

```
int ifft_cplx32x32(int32_t * y, int32_t* x, fft_handle_t h, int scalingOption)
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_cplx16x16(int16_t * y, int16_t* x, fft_handle_t h, int scalingOption)
```

FFT handles:

N	32x32	24x24	32x16	16x16
16	cifft32_16	cifft24_16	cifft16_16	cifft16x16_16
32	cifft32_32	cifft24_32	cifft16_32	cifft16x16_32
64	cifft32_64	cifft24_64	cifft16_64	cifft16x16_64
128	cifft32_128	cifft24_128	cifft16_128	cifft16x16_128
256	cifft32_256	cifft24_256	cifft16_256	cifft16x16_256
512	cifft32_512	cifft24_512	cifft16_512	cifft16x16_512
1024	cifft32_1024	cifft24_1024	cifft16_1024	cifft16x16_1024
2048	cifft32_2048	cifft24_2048	cifft16_2048	cifft16x16_2048
4096	cifft32_4096	cifft24_4096	cifft16_4096	cifft16x16_4096

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 2.5)
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

2.5.4 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`.

Precision

4 versions available:

Type	Description
32x32	32-bit input/outputs, 32-bit twiddles
24x24	24-bit input/outputs, 24-bit twiddles
32x16	32-bit input/outputs, 16-bit twiddles
16x16	16-bit input/outputs, 16-bit twiddles

Algorithm

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

Prototype

```
int ifft_real32x32(int32_t* y, int32_t* x, fft_handle_t h, int scalingOpt)
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_real16x16(int16_t* y, int16_t* x, fft_handle_t h, int scalingOpt)
```

FFT handles:

N	32x32	24x24	32x16	16x16
32	rifft32_32	rifft24_32	rifft16_32	rifft16x16_32
64	rifft32_64	rifft24_64	rifft16_64	rifft16x16_64
128	rifft32_128	rifft24_128	rifft16_128	rifft16x16_128
256	rifft32_256	rifft24_256	rifft16_256	rifft16x16_256
512	rifft32_512	rifft24_512	rifft16_512	rifft16x16_512
1024	rifft32_1024	rifft24_1024	rifft16_1024	rifft16x16_1024
2048	rifft32_2048	rifft24_2048	rifft16_2048	rifft16x16_2048
4096	rifft32_4096	rifft24_4096	rifft16_4096	rifft16x16_4096
8192	rifft32_8192	rifft24_8192	rifft16_8192	rifft16x16_8192

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 2.5)
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

2.5.5 Discrete Cosine Transform

Description

These functions apply DCT (Type II) to input
NOTES:
1. DCT runs in-place algorithm, so input data will appear damaged after the call.

Precision

5 versions available:

Type	Description
32x32	32-bit input/outputs, 32-bit twiddles
24x24	24-bit input/outputs, 24-bit twiddles
32x16	32-bit input/outputs, 16-bit twiddles
16x16	16-bit input/outputs, 16-bit twiddles
f	floating point

Algorithm

$$y = DCT(x)$$

Prototype

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

Arguments

Type	Name	Size	Description
Input			
`int32_t, f24, int16_t, float32_t`	`x`	`N`	input signal
`int`	`N`		DCT size (32 for fixed point functions, 32 or 64 for floating point function)
`int`	`scalingOpt`		scaling option (see table in para 2.5), not applicable to the floating point function
Output			
`int32_t, f24, int16_t, float32_t`	`y`	`N`	output of transform

Returned value

total number of right shifts occurred during scaling procedure (always 5 for fixed point functions and 0 for floating point function)

Restrictions

`x,y` should not overlap
`x,y` - aligned on 8-bytes boundary
`N` - 32 for fixed point functions, 32 or 64 for floating point functions.

2.5.6 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

3 versions available:

Type	Description
`24x24`	24-bit input/outputs, 24-bit twiddles
`32x16`	32-bit input/outputs, 16-bit twiddles
`f`	floating point

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_cplxf_ie (
             complex_float * y, complex_float * x,
             const complex_float* twd,
             int twdstep, int N );
```

Arguments

Type	Name	Size	Description
Input			
`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 2.5) , not applicable to the floating point function
Output			
`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 8-byte boundary

2.5.7 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

5 versions available:

Type	Description
`24x24`	24-bit input/outputs, 24-bit twiddles
`32x16`	32-bit input/outputs, 16-bit twiddles
`24x24_ie_24p`	24-bit packed input/outputs, 24-bit data, 24-bit twiddles
`32x16_ie_24p`	24-bit packed input/outputs, 32-bit data, 16-bit twiddles
`f`	floating point

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_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				
`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 2.5) , not applicable to the floating point function
Output				
`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

2.5.8 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

3 versions available:

Type	Description
24x24	24-bit input/outputs, 24-bit twiddles
32x16	32-bit input/outputs, 16-bit twiddles
f	floating point

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_cplxf_ie(
              complex_float* y, complex_float * x,
              const complex_float * twd,
              int twdstep, int N);
```

Arguments

Type	Name	Size	Description
Input			
complex_fract32, 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 2.5) , not applicable to the floating point function
Output			
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

2.5.9 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

5 versions available:

Type	Description
`24x24`	24-bit input/outputs, 24-bit twiddles
`32x16`	32-bit input/outputs, 16-bit twiddles
`24x24_ie_24p`	24-bit packed input/outputs, 24-bit data, 24-bit twiddles
`32x16_ie_24p`	24-bit packed input/outputs, 32-bit data, 16-bit twiddles
`f`	floating point

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_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_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 2.5) , 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

2.6 Identification Routines

2.6.1 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 point to a buffer large enough to hold up to 30 characters

2.6.2 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

3 Test Environment and Examples

3.1 Supported Use Environment, Configurations and Targets

NatureDSP library and corresponding testdriver supported to be built and tested using Xtensa Xplorer IDE running under Windows, or Linux operating system.

Library is compatible with HiFi 1 DSP cores having HiFi 1 Vector FPU. Floating-Point kernels require VFPU to be part of the configuration.

3.2 Building the NatureDSP Signal Library and the Testdriver

3.2.1 Importing the workspaces in Xtensa Xplorer

NatureDSP Libraries for HiFi 1 are provided as two workspaces:

- Library workspace `HiFi1_lib.xws`

  This workspace contains `HiFi1_library` project with optimized kernels and modules required for demo workspace.

- Demo workspace `HiFi1_demo.xws`

  This contains the `HiFi1_demo` demo project.

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

Make sure that the library workspace is imported first. This is because the project in the demo 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.

3.2.2 Building and Running Tests

Building the library: In Xtensa Xplorer, select the `HiFi1_library` project, and `Debug` or `Release` target, and build. By default the software will be built with xt-clang compiler.

Building the test bench: In Xtensa Xplorer, select the `HiFi1_demo` project, select Debug or Release target, and build.

To run the test bench, select `HiFi1_demo` project, and Run. This will execute each routine of the `HiFi1_library` in cycles performance (MIPS) mode.

Use `--turbo` as runtime argument to test library for functional correctness

3.2.3 Command-line Options

You may wish to launch a separate test by passing command-line options to the executable:
Executing the testdriver without options performs performance testing of library with vectors `-full`.

Functional testing is performed with the explicit command-line option `-func,` with either `-brief` or `-full` vectors. 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. Performance data is always executed with `-mips -full`. There is no `-mips -brief` option.

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

`-help or -h`	List of available options
`-mips`	Performance test
`-func`	Functional test
`-full`	Use full test vector set (if available in the directory `vectors_full`) for deeper validation
`-brief`	Use brief vector set (if available in the directory `vectors_brief`) for functional validation
`-sanity`	Use sanity vector set (if available in the directory `vectors_sanity`) for functional validation
`-dct`	DCT tests
`-fft`	FFT tests
`-cfft`	Complex floating point FFT and fixed point FFT with memory improved usage tests
`-rfft`	Real floating point FFT and fixed point FFT with memory improved usage tests
`-fir`	FIR tests
`-iir`	IIR tests
`-vec`	Vector operations
`-math`	Vector mathematics
`-mtx`	Matrix operations
`-mtxinv`	Matrix inversions
`-phase1`	Test fixed-point routines
`-phase2`	Test floating-point routines

3.2.4 Other Supported Environments

The library and testdriver project might be built under following toolchains and operating systems:

OS	Language	Environment	Tool
Linux	C	xt-clang	make
Windows	C	xt-clang	xt-make

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 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_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_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 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_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, 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
```

4.2 Matlab Code for Generating 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

```
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_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));
```

4.3 Accuracy of Mathematical Functions

Some math functions such as sine, cosine, etc. specify accuracy of computation. General approach is to provide accuracy of 2 LSB (+/-2) for 16x16 routines and 2 ULP for single precision floating point routines. The 24x24 and 32x32 routines are normally more accurate than 16x16, but the level of accuracy varies from function to function.
Accuracy for functions is guaranteed over defined domain i.e. for `vec_sinef()` it is 2 ULP for all values in range `[-102940.0, 102940.0]`. For most fixed-point functions, domain is a whole range from minimum to maximum number, for example, `-32768` to `32767` for 16x16 routines.
Since mathematical functions are defined for continuous values while the processor works with discrete numbers, the method of measuring accuracy requires clarification.
Accuracy for function `y=f(x)` is defined as a maximum difference between digitized output of function `y` and its digitized ideal value `f(x)`. Input for function is digitized as well. In practice, seq-files contain input data for function and allowable upper/lower bounds for its output. Test harness tool just passes input data to function under the test and compares output with upper/lower bounds from the seq-file.

As an example, look for the test data from `scl_sine16x16.seq`:

```
13 2; test case # and type (small and negative N)
2 ; N
-9076 -8055 ; x[2]
-25051 -22866 ; ylo[2] (lower bound)
```

```
-25047 -22862 ; yhi[2] (upper bound)
```

Here we have 2 inputs for sine function. Inputs and outputs are in Q15 and input is scaled by pi, so expected outputs might be computed by Matlab code:

```
round(pow2(sin(pow2([-9076 -8055],-15)*pi),15))
```

result is `[-25049, -22864]`.

So, lower and higher bounds differ from expected result by 2.

Similar case from `scl_sinef()`

```
2 1; test case # and type (known data)
1 ; N
47154fc2 ; x[1]
367a1d9c ; ylo[1] (lower bound)
367a1da0 ; yhi[1] (upper bound)
```

Here all the single precision data are shown in their hex machine representation as they appear in the processor.

In natural representation, input is

```
x=(typecast(int32(hex2dec('47154fc2')),'single'))
38223.7578125
```

Right sine value is `3.7270143e-006` or

`num2hex(single(sin(double(x))))=367a1d9e` in machine representation

So, we see that upper/lower bounds in seq-file are 2 ULP apart from this ideal value.

For 24x24, the situation is bit different because 24-bit data are stored in 3 most significant bytes of 32-bit data. So, for input data, lower 8 bits should be ignored, and ideal output should be computed using digitized inputs with cleared 8 LSBs. Comparison with lower/upper bounds should ignore these 8 LSBs, this is achieved by small modification of bounds - 8 LSB for lower bound is cleared and 8 LSB for upper bound are set.

Here is the part of `scl_sine24x24.seq`:

```
22 2; test case # and type (small and negative N)
4 ; N
1519943108 1365826688 2070175853 2062996324 ; x[4]
1705915136 1954328576 242278400 264675584 ; ylo[4] (lower bound)
1706063359 1954476799 242426623 264823807 ; yhi[4] (upper bound)
```

So, for first input `x[0]=1519943108` we have digitized value `1519943108&~255=1519942912`.
The ideal sine value for it is `y=round(pow2(sin(pow2(x,-31)*pi),31))=1705989149`.
Function provides accuracy `74000`, so lower bound is `(y-74000)&~255=1705915136` and upper bound is `(y+74000)|255=1706063359`

Fixed point functions usually indicate decimal accuracy as well as its floating-point equivalent. The relationship between these quantities depends on fractional format which is specific for each function. For example, `vec_sine24x24()` specifies decimal accuracy `74000`. Output is represented in Q31, so floating point accuracy will be `74000*2^-31=3.4e-5`. See chapter 1.4 for details.