NatureDSP Signal

Digital Signal Processing

Library Reference

For Tensilica HiFi2 DSP

Date: 11-June-2010
Table of Contents

PREFACE..........................................................................................................................................................5

ABOUT THIS MANUAL.............................................................................................................................5
SUPPORTED TARGETS.............................................................................................................................5
NOTATIONS.......................................................................................................................................................5
ABBREVIATIONS..............................................................................................................................................5

CHAPTER 1 GENERAL LIBRARY ORGANIZATION .........................................................................................6
1.1 HEADERS.....................................................................................................................................................6
1.2 STATIC VARIABLES AND USAGE OF C STANDARD LIBRARIES..........................................................6
1.3 TYPES..........................................................................................................................................................6
1.4 FRACTIONAL FORMATS ..........................................................................................................................6
1.5 COMPILER REQUIREMENTS......................................................................................................................7
1.6 CALL CONVENTIONS................................................................................................................................7
1.7 OVERFLOW CONTROL.............................................................................................................................7
1.8 ENDIANESS..................................................................................................................................................7
1.9 PERFORMANCE ISSUES............................................................................................................................7
1.10 BRIEF FUNCTION LIST...........................................................................................................................7

CHAPTER 2 REFERENCE ...............................................................................................................................................9
2.1 FIR FILTERS AND RELATED FUNCTIONS ............................................................................................9
2.1.1 Block real FIR filter................................................................................................................................9
2.1.2 Complex block FIR filter ....................................................................................................................9
2.1.3 Symmetrical block real FIR filter .......................................................................................................10
2.1.4 Decimating block real FIR filter.........................................................................................................10
2.1.5 Interpolating block real FIR filter .....................................................................................................11
2.1.6 Single-sample real FIR filter ...........................................................................................................12
2.1.7 Convolution.......................................................................................................................................12
2.1.8 Correlation.......................................................................................................................................13
2.1.9 Autocorrelation................................................................................................................................14
2.1.10 Raw single sample delayed LMS algorithm....................................................................................14
2.1.11 Blockwise Adaptive LMS algorithm...............................................................................................15
2.2 IIR FILTERS............................................................................................................................................16
2.2.1 Bi-quad complex block IIR...............................................................................................................16
2.2.2 Bi-quad double precision IIR filter....................................................................................................17
2.2.3 Lattice block Real IIR......................................................................................................................18
2.2.4 Lattice complex block IIR................................................................................................................19
2.2.5 Lattice double precision IIR filter.....................................................................................................20
2.3 VECTOR MATHEMATICS.........................................................................................................................20
2.3.1 Vector dot product.............................................................................................................................20
2.3.2 Vector sum.......................................................................................................................................21
2.3.3 Power of a vector................................................................................................................................21
2.3.4 Vector scaling with saturation..........................................................................................................22
2.3.5 Reciprocal on Q31 numbers.............................................................................................................22
2.3.6 Division of Q31 numbers..................................................................................................................23
2.3.7 Logarithm.......................................................................................................................................24
2.3.8 Sine/cosine.......................................................................................................................................25
2.3.9 Tangent.............................................................................................................................................25
2.3.10 Full arctangent...............................................................................................................................26
2.3.11 Arctangent....................................................................................................................................27
2.3.12 Common exponent.........................................................................................................................27
2.4 MATRIX OPERATIONS ............................................................................................................................28
2.4.1 Matrix multiply..................................................................................................................................28
2.4.2 Matrix transpose.............................................................................................................................29
2.5 FAST FOURIER TRANSFORMS..................................................................................................................29
2.5.1 FFT on complex data........................................................................................................................29
2.5.2 FFT on real data..............................................................................................................................30
2.5.3 Inverse FFT on complex data .........................................................................................................30
2.5.4 Inverse FFT forming real data .......................................................................................................31

Copyright 2009-2010, IntegrIT Ltd., www.integrit.ru
CHAPTER 3  PERFORMANCE MEASUREMENTS ........................................................................... 33

3.1  FUNCTIONS PERFORMANCE LIST .................................................................................. 33

CHAPTER 4  TEST ENVIRONMENT AND EXAMPLES ................................................................... 35

4.1  HOW TO BUILD AND RUN TESTS ..................................................................................... 35
   4.1.1  Building the NatureDSP Signal library ........................................................................ 35
   4.1.2  Building and running tests .......................................................................................... 35

4.2  FIR TESTS .......................................................................................................................... 37

4.3  MATRIX OPERATIONS TESTS ........................................................................................... 37

4.4  VECTOR OPERATIONS TESTS ............................................................................................ 38

4.5  IIR TESTS ........................................................................................................................... 38

4.6  FFT TESTS .......................................................................................................................... 38

CHAPTER 5  CUSTOMER SUPPORT ........................................................................................... 39
Preface

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

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

Supported targets
Library supports Tensilica ConnX D2 and HiFi2 targets. Call IntegrIT to support more targets.

Notations
This document uses the following conventions:
- program listings, program examples, interactive displays, filenames, variables and another software elements are shown in a special typeface (Courier);
- tables use smaller fonts.

Abbreviations
4FSK 4-level frequency shift keying
API Application program interface
DSP Digital signal processing
FFT Fast Fourier transform
FIR Finite impulse response
IDE Integrated development environment
IFFT Inverse Fast Fourier transform
IIR Infinite impulse response
IR Impulse response
LMS Least mean squares
Chapter 1 General library organization

1.1 Headers

The `NatureDSP_Signal` library is supplied with a number of header files:

<table>
<thead>
<tr>
<th>Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;top&gt;/NatureDSP_types.h</td>
<td>Declarations of basic data types and compiler autodetection</td>
</tr>
<tr>
<td>NatureDSP_Signal.h</td>
<td>Declarations of library functions</td>
</tr>
</tbody>
</table>

1.2 Static variables and usage of C standard libraries

All library functions are re-entrant. Library does not require floating point support from the compiler or toolchain.

1.3 Types

The library uses the following C types with defined length:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f24</code></td>
<td>24-bit fractional type</td>
</tr>
<tr>
<td><code>int32</code></td>
<td>32-bit signed value</td>
</tr>
<tr>
<td><code>i56</code></td>
<td>Special 56-bit formatted data packed to 64-bit words</td>
</tr>
</tbody>
</table>

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

1.4 Fractional formats

Natively, HiFi2 CPU uses special fractional type `f24` which is stored in a memory as 32-bit word keeping significant bits in bits 8 through 31. So, from that perspective it may be treated as `Q31` number. But users should take into account that 8 LSB are ignored. **Unless specifically noted, library functions use that `Q31` format, or, in another words, `Q0.31`**.

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

<table>
<thead>
<tr>
<th>Format</th>
<th>Range</th>
<th>Minimum value</th>
<th>Maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q6.15</td>
<td>-1 ... 0,999969</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>Q6.9</td>
<td>-64 ... 63,998</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>Q0.30</td>
<td>-2 ... 1,9999999991</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td>Q1.30</td>
<td>-1 ... 0,9999999995</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
</tbody>
</table>

The most-significant binary digit is interpreted as the sign bit in any `Q` format number. Thus, in `Q15` format, the decimal point is placed immediately to the right of the sign bit. The fractional portion to the right of the sign bit is stored in regular two’s complement format.
1.5 Compiler requirements
When building the library source files or library-dependent modules it is assumed that the target is a Tensilica processor implementing the Xtensa HiFi2 Audio Engine Instruction Set Architecture.

1.6 Call conventions
Library uses ANSI-C call conventions.

1.7 Overflow control
If not especially noted, library does not check real dynamic range of input data so it is user’s responsibility to select parameters and the scale of input data according to specific case. However, if possible library use saturated arithmetic to prevent overflows. The user is expected to conform to the range requirements if specified and take care to restrict the input range in such a way that the outputs do not overflow.

1.8 Endianness
Library supports both little and big endian mode.

1.9 Performance issues
Real-time performance of all functions depends on fulfillment special restrictions applied to input/output arguments. Typically, for maximum performance, user have to use **aligned data arrays (on 8 byte boundary)** for storing input and output arguments, number of data should be **divisible by 2 or 4** and should be **greater than 4**. Specific requirements are given for each function in its API description. Data alignment may be achieved by several methods:
- placing the data into special data section and make alignment at the link-time
- use _declspec(align(x)) modifiers in the data declarations
- dynamically allocate arrays of lighter bigger size and align pointers
Test examples use two last methods.

1.10 Brief function list

<table>
<thead>
<tr>
<th>Vectorized version</th>
<th>Scalar version</th>
<th>Purpose</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FIR filters and related functions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fir_bk</td>
<td>Block real FIR filter</td>
<td>2.1.1</td>
<td></td>
</tr>
<tr>
<td>fir_ckb</td>
<td>Complex block FIR filter</td>
<td>2.1.2</td>
<td></td>
</tr>
<tr>
<td>fir_sr</td>
<td>Symmetrical block real FIR filter</td>
<td>2.1.3</td>
<td></td>
</tr>
<tr>
<td>fir_dec</td>
<td>Decimating block real FIR filter</td>
<td>2.1.4</td>
<td></td>
</tr>
<tr>
<td>fir_interp</td>
<td>Interpolating block real FIR filter</td>
<td>2.1.5</td>
<td></td>
</tr>
<tr>
<td>fir_s</td>
<td>Single-sample real FIR filter.</td>
<td>2.1.6</td>
<td></td>
</tr>
<tr>
<td>fir_conv</td>
<td>Convolution</td>
<td>2.1.7</td>
<td></td>
</tr>
<tr>
<td>fir_xcorr</td>
<td>Correlation</td>
<td>2.1.8</td>
<td></td>
</tr>
<tr>
<td>fir_acorr</td>
<td>Autocorrelation</td>
<td>2.1.9</td>
<td></td>
</tr>
<tr>
<td>fir_dms</td>
<td>Raw single sample delayed LMS algorithm</td>
<td>2.1.10</td>
<td></td>
</tr>
<tr>
<td>fir_dms</td>
<td>Blockwise Adaptive LMS algorithm</td>
<td>2.1.11</td>
<td></td>
</tr>
<tr>
<td><strong>IIR filters</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iir_bqc</td>
<td>Biquad Complex block IIR</td>
<td>2.2.1</td>
<td></td>
</tr>
<tr>
<td>iir_bqd</td>
<td>Biquad Double precision IIR filter</td>
<td>2.2.2</td>
<td></td>
</tr>
<tr>
<td>iir_latr</td>
<td>Lattice block Real IIR</td>
<td>2.2.3</td>
<td></td>
</tr>
<tr>
<td>iir_lattc</td>
<td>Lattice complex block IIR</td>
<td>2.2.4</td>
<td></td>
</tr>
<tr>
<td>iir_lattd</td>
<td>Lattice double precision IIR filter</td>
<td>2.2.5</td>
<td></td>
</tr>
<tr>
<td><strong>Vector mathematics</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vectorized version</td>
<td>Scalar version</td>
<td>Purpose</td>
<td>Reference</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------</td>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td>vec_dot</td>
<td></td>
<td>Vector dot product</td>
<td>2.3.1</td>
</tr>
<tr>
<td>vec_add</td>
<td></td>
<td>Vector sum</td>
<td>2.3.2</td>
</tr>
<tr>
<td>vec_power</td>
<td></td>
<td>Power of a vector</td>
<td>2.3.3</td>
</tr>
<tr>
<td>vec_shift</td>
<td></td>
<td>Vector scaling with saturation</td>
<td>2.3.4</td>
</tr>
<tr>
<td>vec_recip24</td>
<td>scl_recip24</td>
<td>Reciprocal on a vector of Q31 numbers</td>
<td>2.3.5</td>
</tr>
<tr>
<td>vec_divide</td>
<td>scl_divide</td>
<td>Division</td>
<td>2.3.6</td>
</tr>
<tr>
<td>vec_logn</td>
<td>scl_logn</td>
<td>Different kinds of logarithm</td>
<td>2.3.7</td>
</tr>
<tr>
<td>vec_log2</td>
<td>scl_log2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vec_log10</td>
<td>scl_log10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vec_sine</td>
<td>scl_sine</td>
<td>Sine</td>
<td>2.3.8</td>
</tr>
<tr>
<td>vec_cosine</td>
<td>scl_cosine</td>
<td>Cosine</td>
<td></td>
</tr>
<tr>
<td>vec_tan</td>
<td>scl_tan</td>
<td>Tangent</td>
<td>2.3.9</td>
</tr>
<tr>
<td>vec_atan2</td>
<td>scl_atan2</td>
<td>Full arctangent</td>
<td>2.3.10</td>
</tr>
<tr>
<td>vec_atan</td>
<td>scl_atan</td>
<td>arctangent</td>
<td>2.3.11</td>
</tr>
<tr>
<td>vec_bexp</td>
<td>scl_bexp</td>
<td>Common exponent</td>
<td>2.3.12</td>
</tr>
</tbody>
</table>

Matrix operations

| mtx_mpy           | Matrix multiply | 2.4.1     |
| mtx_trans         | Matrix transpose | 2.4.2     |

FFT

| fft_cplx          | FFT on complex data | 2.5.1     |
| fft_real          | FFT on real data   | 2.5.2     |
| ifft_cplx         | Inverse FFT on complex data | 2.5.3     |
| ifft_real         | Inverse FFT forming real data | 2.5.4     |
Chapter 2  Reference

2.1 FIR filters and related functions

2.1.1 Block real FIR filter

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

Algorithm
\[ y_n = \sum_{m=0}^{M-1} h_{m-1-m} x_{n+m}, \quad n = 0 \ldots N-1 \]

Prototype

```c
void fir_bk (      f24 * restrict y,
const f24 * restrict x,
const f24 * restrict h,
int N,
int M)
```

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f24</td>
<td>( x )</td>
<td>( N+M-1 )</td>
<td>input data, First in time corresponds to ( x[0] )</td>
</tr>
<tr>
<td>f24</td>
<td>( h )</td>
<td>( M )</td>
<td>filter coefficients; ( h[0] ) is to be multiplied with the newest sample</td>
</tr>
<tr>
<td>int</td>
<td>( N )</td>
<td></td>
<td>length of sample block</td>
</tr>
<tr>
<td>int</td>
<td>( M )</td>
<td></td>
<td>length of filter</td>
</tr>
</tbody>
</table>

Output:

| f24   | \( y \) | \( N \) | output data                                      |

Returned value
None

Restrictions
\( x, h, y \) should not overlap

Performance restrictions:
\( N, M \) - multiples of 4 exceeding 8

Code/Data footprint:
0x273 (.text)

Cycle count example:
Invocation parameters: \( N \): 80; \( M \): 256; \( h \): aligned on a 8-bytes boundary
Cycles count: 12437

2.1.2 Complex block FIR filter

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

Algorithm
\[ y_n = \sum_{m=0}^{M-1} h_{m-1-m} x_{n+m}, \quad n = 0 \ldots N-1 \]

Prototype

```c
void fir_cbk (      f24 * restrict y,
const f24 * restrict x,
const f24 * restrict h,
int N,
int M)
```

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f24</td>
<td>( x )</td>
<td>( 2 \times (N+M-1) )</td>
<td>complex input data. First in time corresponds to ( x[0] )</td>
</tr>
</tbody>
</table>
| f24   | \( h \) | \( 2 \times M \)  | complex filter coefficients, \( h[0], h[1] \) is to be }
multiplied with the newest sample

\[
\text{int } N \quad \text{length of sample block}
\]

\[
\text{int } M \quad \text{length of filter}
\]

\text{Output:}

\[
\text{f24 } t \quad 2^*N \quad \text{complex output data}
\]

\text{Returned value none}

\text{Restrictions} \quad x, h, y \text{ should not overlap}

\text{Performance restrictions:}

\[
N \text{ } \text{a multiple of 2}
\]

\[
M \text{ } \text{greater than eight}
\]

\text{Code/Data footprint:}

0x237 (.text)

\text{Cycle count example:}

Invocation parameters: N: 80; M: 128; h: aligned on a 8-bytes boundary

Cycles count: 22013

### 2.1.3 Symmetrical block real FIR filter

**Description**

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

**Algorithm**

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

**Prototype**

\[
\text{void fir_sr ( f24 * restrict y, const f24 * restrict x, const f24 * restrict h, int N, int M)}
\]

**Arguments**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>( x )</td>
<td>( N+M-1 )</td>
<td>input data. First in time corresponds to ( x[0] )</td>
</tr>
<tr>
<td>f24</td>
<td>( h )</td>
<td>( (N+1)/2 )</td>
<td>A half of filter coefficients. Last tap suggested to be a center of IR.</td>
</tr>
<tr>
<td>int</td>
<td>( N )</td>
<td></td>
<td>length of sample block</td>
</tr>
<tr>
<td>int</td>
<td>( M )</td>
<td></td>
<td>length of filter</td>
</tr>
</tbody>
</table>

**Output:**

\[
f24 \quad t \quad N \quad \text{output data}
\]

\text{Returned value none}

\text{Restrictions} \quad x, h, y \text{ should not overlap}

\text{Performance restrictions:}

\[
x, h \text{ } \text{aligned on a 8-bytes boundary}
\]

\[
y \text{ } \text{a multiple of 4}
\]

\text{Code/Data footprint:}

0x5de (.text)

\text{Cycle count example:}

Invocation parameters: N: 80; M: 255; h: aligned on a 8-bytes boundary

Cycles count: 13433

### 2.1.4 Decimating block real FIR filter

**Description**

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

**NOTE:**

To avoid aliasing IR should be synthesized in such a way to be narrower than input sample rate divided to \( 2D \).
Algorithm
\[ r_n = \sum_{m=0}^{M-1} h_{M-1-m} x_{Dn+m}, n = 0...N-1 \]

Prototype
```c
void fir_dec (    f24 * restrict r,
                const f24 * restrict x,
                const f24 * restrict h,
                int N,
                int M,
                int D)
```

Arguments
<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f24</td>
<td>x</td>
<td>D*N+M-1</td>
<td>input data. First in time corresponds to x[0]</td>
</tr>
<tr>
<td>f24</td>
<td>h</td>
<td>M</td>
<td>filter coefficients; h[0] is to be multiplied with the newest sample</td>
</tr>
<tr>
<td>int</td>
<td>N</td>
<td></td>
<td>length of sample block</td>
</tr>
<tr>
<td>int</td>
<td>M</td>
<td></td>
<td>length of filter</td>
</tr>
<tr>
<td>int</td>
<td>D</td>
<td></td>
<td>decimation factor</td>
</tr>
<tr>
<td>Output:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f24</td>
<td>r</td>
<td>N</td>
<td>output data</td>
</tr>
</tbody>
</table>

Returned value
none

Restrictions
| x, h, r should not overlap |
| D should exceed 1 |

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

Code/Data footprint:
0x4ad (.text)

Cycle count example:
Invocation parameters: N: 80; M: 256; D: 2; h: aligned on a 8-bytes boundary
Cycles count: 11196

2.1.5 Interpolating block real FIR filter

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

Function requires special storage format of IR. Original IR should be decimated by \( D \) with offsets \( 0..D-1 \). Resulting \( D \) subfilters should be stored sequentially. For each subfilter the coefficient that corresponds to the newest sample is to be stored first.

Algorithm
\[ y_{n,D,d} = \sum_{m=0}^{M-1} h_{(d+1)M-1-m} x_{n+Dm}, n = 0...N-1, d = 0...D-1, \]

Prototype
```c
void fir_interp (f24 * restrict y,
                 const f24 * restrict x,
                 const f24 * restrict h,
                 int N,
                 int M,
                 int D)
```

Arguments
<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f24</td>
<td>x</td>
<td>N+M*D-1</td>
<td>input data. First in time corresponds to x[0]</td>
</tr>
<tr>
<td>f24</td>
<td>h</td>
<td>M*D</td>
<td>filter coefficients</td>
</tr>
<tr>
<td>int</td>
<td>N</td>
<td></td>
<td>length of sample block</td>
</tr>
<tr>
<td>int</td>
<td>M</td>
<td></td>
<td>length of subfilter</td>
</tr>
<tr>
<td>int</td>
<td>D</td>
<td></td>
<td>interpolation (upsample) factor</td>
</tr>
<tr>
<td>Output:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f24</td>
<td>y</td>
<td>N*D</td>
<td>output data</td>
</tr>
</tbody>
</table>

Returned value
none

Restrictions
| x, h, y should not overlap |
| D should be >1 |
2.1.6 Single-sample real FIR filter

Description

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

NOTE:
The difference with `fir_bk()` function is that this function returns sample in Q30, not Q31 format.

Algorithm

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

Prototype

```c
f24 fir_ss ( f24 * restrict x,
             const f24 * restrict h,
             int M)
```

Arguments

- **Type**: `f24`
- **Name**: `x`
- **Size**: `M`
- **Description**: Input data. First in time corresponds to $x[0]$. User should place new arrived sample to the last position of $x$.

- **Type**: `f24`
- **Name**: `h`
- **Size**: `M`
- **Description**: Filter coefficients in normal order; $h[0]$ is to be multiplied with the newest sample.

- **Type**: `int`
- **Name**: `M`
- **Description**: Length of filter.

Output:

- **Type**: `f24`
- **Name**: `r`
- **Description**: Filtered data, Q30

Returns

- **Type**: `f24`
- **Name**: `r`
- **Description**: Filtered data, Q30

Restrictions

$x, h$ should not overlap.

Performance restrictions:

- $x$ - aligned on a 8-bytes boundary
- $h$ - a multiple of 4 exceeding 8

Code/Data footprint:

0x19d (.text)

Cycle count example:

Invocation parameters: $M$: 256; $h$: aligned on a 8-bytes boundary
Cycles count: 443

2.1.7 Convolution

Description

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

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

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

Algorithm

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

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

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

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

```c
void fir_convol ( f24 * restrict r,
    const f24 * restrict x,
    const f24 * restrict y,
    int N,
    int M)
```

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>x</td>
<td>N</td>
<td>input data, Q31</td>
</tr>
<tr>
<td>f24</td>
<td>y</td>
<td>M</td>
<td>input data, Q31</td>
</tr>
<tr>
<td>int</td>
<td>N</td>
<td></td>
<td>length of x</td>
</tr>
<tr>
<td>int</td>
<td>M</td>
<td></td>
<td>length of y</td>
</tr>
</tbody>
</table>

Output:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32</td>
<td>r</td>
<td>M+N-1</td>
<td>output data, Q31</td>
</tr>
</tbody>
</table>

Returned value: none

Restrictions:

- x, y, r should not overlap
- x, y aligned on a 8-bytes boundary
- N, M multiples of 4 exceeding 8

Performance restrictions:

- x, y aligned on a 8-bytes boundary
- N, M multiples of 4 exceeding 8

Code/Data footprint:

- 0x5c1 (.text)

Cycle count example:

- Invocation parameters: N: 80; M: 56; x, y aligned on a 8-bytes boundary
- Cycles count: 4030

2.1.8 Correlation

Description

Estimates the cross-correlation between vectors x and y.

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

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

Algorithm

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

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

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

Prototype

```c
void fir_xcorr ( f24 * restrict r,
    const f24 * restrict x,
    const f24 * restrict y,
    int N,
    int M)
```

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>x</td>
<td>N</td>
<td>input data</td>
</tr>
<tr>
<td>f24</td>
<td>y</td>
<td>M</td>
<td>input data</td>
</tr>
<tr>
<td>int</td>
<td>N</td>
<td></td>
<td>length of x</td>
</tr>
<tr>
<td>int</td>
<td>M</td>
<td></td>
<td>length of y</td>
</tr>
</tbody>
</table>

Output:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>r</td>
<td>M+N-1</td>
<td>output data</td>
</tr>
</tbody>
</table>

Returned value: none

Restrictions:

- x, y, r should not overlap
- x, y aligned on a 8-bytes boundary
- N, M multiples of 4 exceeding 8

Performance restrictions:

- x, y aligned on a 8-bytes boundary
- N, M multiples of 4 exceeding 8

Code/Data footprint:

- 0x733 (.text)

Cycle count example:

- Invocation parameters: N: 80; M: 56; x, y aligned on a 8-bytes boundary
- Cycles count: 4041
2.1.9 Autocorrelation

**Description**
Estimates the auto-correlation between of vector \( x \). Returns autocorrelation in a length \( N \) vector, where \( x \) is length \( N \) vector.

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

**Algorithm**
\[
r_k = \sum_{n=0}^{N-1-k} x_{n+k} x_n, k = 0... (N-1)
\]

**Prototype**

```
void fir_acorr ( f24 * restrict r, const f24 * restrict x, int N);
```

**Arguments**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>f24</td>
<td>( x )</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>f24</td>
<td>( N )</td>
<td>length of ( x )</td>
</tr>
<tr>
<td>Output:</td>
<td>f24</td>
<td>( r )</td>
<td>N</td>
</tr>
</tbody>
</table>

**Returned value**
one

**Restrictions**
\( x, r \) should not overlap

**Performance restrictions:**
\( x \) - aligned on a 8-bytes boundary
\( N \) - a multiple of 4 exceeding 8

**Code/Data footprint:**
0x327 (.text)

**Cycle count example:**
Invocation parameters: \( N: 56; x \) : aligned on a 8-bytes boundary
Cycles Count: 1515

2.1.10 Raw single sample delayed LMS algorithm

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

**Algorithm**
\[
r = \sum_{m=0}^{N-1} h_{N-1-m} x_{m+1}
\]
\[
h_{N-1-m} = h_{N-1-m} + b \cdot x_m, m = 0...N-1
\]

**Prototype**

```
f24 fir_dlms ( f24 * restrict h, const f24 * restrict x, f24 b, int N);
```

**Arguments**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>f24</td>
<td>( h )</td>
<td>( N )</td>
</tr>
<tr>
<td></td>
<td>f24</td>
<td>( x )</td>
<td>N+1</td>
</tr>
<tr>
<td></td>
<td>f24</td>
<td>( b )</td>
<td>scaled error from previous step (Q32). Normally, is formed as a difference of a learning sequence and the filter output. Scaling has to be done to take into account needed LMS step.</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>( N )</td>
<td>length of ( h )</td>
</tr>
</tbody>
</table>

**Output:**
2.1.11 Blockwise Adaptive LMS algorithm

Description
Blockwise LMS algorithm performs filtering of input samples, computation of error over a block of reference samples and makes blockwise update of IR to minimize the error output.
Algorithm includes FIR filtering, calculation of correlation between the error output and reference signal and IR taps update based on that correlation.
NOTE: this algorithm consumes less CPU cycles per block than single sample algorithm at similar convergence rate.

Algorithm
\[
b = \frac{\mu}{\text{norm}}
\]
\[
e_n = r_n - \sum_{m=0}^{M-1} h_{M-1-m} x_{n+m}, \quad n = 0 \ldots N-1
\]
\[
h_{M-1-m} = h_{M-1-m} + b \cdot \sum_{n=0}^{N-1} e_n x_{n+m}, \quad m = 0 \ldots M-1
\]

Prototype
```c
void fir_blms ( f24 * restrict e, f24 * restrict h, const f24 * restrict r, const f24 * restrict x, f24 norm, f24 mu, int N, int M);
```

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f24</td>
<td>h</td>
<td>M</td>
<td>impulse response, Q31</td>
</tr>
<tr>
<td>f24</td>
<td>r</td>
<td>N</td>
<td>reference (near end) data vector. First in time value is in r[0]</td>
</tr>
<tr>
<td>f24</td>
<td>x</td>
<td>N+M-1</td>
<td>input (far end) data vector. First in time value is in x[0]</td>
</tr>
<tr>
<td>f24</td>
<td>norm</td>
<td></td>
<td>normalization factor: power of signal multiplied by N, Q31</td>
</tr>
<tr>
<td>f24</td>
<td>mu</td>
<td></td>
<td>adaptation coefficient in Q31 (LMS step)</td>
</tr>
<tr>
<td>int</td>
<td>N</td>
<td></td>
<td>length of data block</td>
</tr>
<tr>
<td>int</td>
<td>M</td>
<td></td>
<td>length of h</td>
</tr>
<tr>
<td>Output:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f24</td>
<td>e</td>
<td>N</td>
<td>estimated error</td>
</tr>
<tr>
<td>f24</td>
<td>h</td>
<td>M</td>
<td>updated impulse response, Q31</td>
</tr>
</tbody>
</table>

Returned value
none

Restrictions
h, x, r, y, e - should not overlap

Performance restrictions:
x, e, h, r - aligned on a 8-bytes boundary
N, M - multiples of 4 exceeding 8

Code/Data footprint:
0x4b8 (.text)
Cycle count example:
Invocation parameters: N: 64; M: 16; x,e,h,r: aligned on a 8-bytes boundary
Cycles count: 2151

2.2 IIR filters

2.2.1 Bi-quad complex block IIR

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

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

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

Prototype

```c
void iir_bq ( f24 * restrict r,
               f24 * restrict d,
               const f24 * restrict x,
               const f24 * restrict coef,
               int32 gain,
               int N, int M)
```

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f24</td>
<td>d</td>
<td>4*M</td>
<td>complex delay line elements from previous call. Should be zeroed prior to the first call.</td>
</tr>
<tr>
<td>f24</td>
<td>x</td>
<td>2*N</td>
<td>complex input data. First in time corresponds to ( x[0] ). Real and imaginary parts are interleaved and real part goes first.</td>
</tr>
</tbody>
</table>
| f24  | coef | M*6  | filter coefficients stored in blocks of 6 numbers: \( g \ b0 \ b1 \ b2 \ a1 \ a2 \). Fractional formats used:
|       |      |      | \( b0 \ b1 \ b2 \ a1 \ a2 \) \( Q_{30} \)
|       |      |      | \( g \) \( Q_{31} \)
| int32 | gain |      | total gain, \( Q_{8.24} \) |
| int   | N    |      | length of input vector (number of complex data points) |
| int   | N    |      | number of biquads |
### 2.2.2 Bi-quad double precision IIR filter

**Description**

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

**NOTE:**

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

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

**Algorithm**

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

**Prototype**

```c
void iir_bqdf (f24 * restrict r,
               f24 * restrict d,
               const f24 * restrict x,
               const f24 * restrict coef,
               int32 gain,
               int N, int M)
```

**Arguments**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>d</td>
<td>2*M</td>
<td>delay line elements from previous call. Should be zeroed prior to the first call</td>
</tr>
<tr>
<td>f24</td>
<td>x</td>
<td>N</td>
<td>input data. First in time corresponds to ( x[0] )</td>
</tr>
<tr>
<td>f24</td>
<td>coef</td>
<td>M*6</td>
<td>filter coefficients stored in blocks of 6</td>
</tr>
</tbody>
</table>
2.2.3 Lattice block Real IIR

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

Algorithm
Algorithm consists of applying sequentially \( M \) times IIR sections with structure shown below:

![Diagram of IIR filter](image)

Prototype
```c
void iir_latr(
    f24 *restrict   r,
    f24 *restrict   b,
    const f24 * restrict x,
    const f24 * restrict refl,
    int N, int M
)
```

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int16_t</td>
<td>b</td>
<td>M+1</td>
<td>delay line elements from previous call. Should be zeroed prior to the first call</td>
</tr>
<tr>
<td>int16_t</td>
<td>x</td>
<td>N</td>
<td>input data. First in time corresponds to ( x[0] )</td>
</tr>
<tr>
<td>int16_t</td>
<td>refl</td>
<td>M</td>
<td>reflection coefficients</td>
</tr>
<tr>
<td>int</td>
<td>N</td>
<td></td>
<td>length of sample block</td>
</tr>
<tr>
<td>int</td>
<td>M</td>
<td></td>
<td>number of sections</td>
</tr>
</tbody>
</table>

Output:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int16_t</td>
<td>r</td>
<td>N</td>
<td>output data</td>
</tr>
<tr>
<td>int16_t</td>
<td>b</td>
<td>M+1</td>
<td>updated delay line</td>
</tr>
</tbody>
</table>

Returned value
none

Restrictions
\( x, r, b, refl \) should not overlap

Performance restrictions:
\( x, r, b, refl \) - aligned on 8-byte boundary

\( M \) - from the range 1...6
2.2.4 Lattice complex block IIR

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

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

```
\[ \begin{array}{cccc}
\cdots & k_0 & k_1 & \cdots \\
& & & \\
& & & \\
\end{array} \]
```

**Prototype**
```c
void iir_latc(
    f24 * r,
    f24 * restrict b,
    const f24 * restrict x,
    const f24 * restrict refl,
    int N, int M
)
```

**Arguments**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>b</td>
<td>2*(M+1)</td>
<td>complex delay line elements from previous call. Should be zeroed prior to the first call</td>
</tr>
<tr>
<td>f24</td>
<td>x</td>
<td>2*N</td>
<td>complex input data. First in time corresponds to $x[0]$. Real and imaginary parts are interleaved and real part goes first.</td>
</tr>
<tr>
<td>f24</td>
<td>refl</td>
<td>M</td>
<td>reflection coefficients</td>
</tr>
<tr>
<td>int</td>
<td>N</td>
<td></td>
<td>length of sample block</td>
</tr>
<tr>
<td>int</td>
<td>M</td>
<td></td>
<td>number of sections</td>
</tr>
<tr>
<td>f24</td>
<td>r</td>
<td>2*N</td>
<td>complex output data. Real and imaginary parts are interleaved and real part goes first.</td>
</tr>
<tr>
<td>f24</td>
<td>b</td>
<td>2*(M+1)</td>
<td>updated delay line</td>
</tr>
</tbody>
</table>

**Returned value**
none

**Restrictions**
$b, x, refl, r$ should not overlap
$x, r, b, refl$ - aligned on 8-byte boundary
$M$ - from the range 1...6

**Code/ Data footprint:**
0x1077 (.text)

**Cycle count example:**
Invocation parameters: N: 80; M: 6; x,d,b,refl - aligned on a 8-bytes boundary
Cycles count: 3764
2.2.5 Lattice double precision IIR filter

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

In contrast to \texttt{iir_latr()} it performs some computations in double precision and stores delay line elements with extended.

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

```
\[ z^{-1} + k_0 \]
\[ z^{-1} + k_1 \]
\[ \vdots \]
\[ z^{-1} + k_{M-1} \]
```

Prototype

```c
void iir_lattd( f24 *r, i56 *restrict b, const f24 * restrict x, const f24 * restrict refl, int N, int M )
```

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i56</td>
<td>b</td>
<td>M+1</td>
<td>delay line elements from previous call. Should be zeroed prior to the first call</td>
</tr>
<tr>
<td>f24</td>
<td>x</td>
<td>N</td>
<td>input data. First in time corresponds to ( x[0] )</td>
</tr>
<tr>
<td>f24</td>
<td>refl</td>
<td>M</td>
<td>reflection coefficients</td>
</tr>
<tr>
<td>int</td>
<td>N</td>
<td></td>
<td>length of sample block</td>
</tr>
<tr>
<td>int</td>
<td>M</td>
<td></td>
<td>number of sections</td>
</tr>
</tbody>
</table>

Output:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>r</td>
<td>N</td>
<td>output data</td>
</tr>
<tr>
<td>i56</td>
<td>b</td>
<td>M+1</td>
<td>updated delay line</td>
</tr>
</tbody>
</table>

Returned value none

Restrictions
\( b, x, refl, r \) should not overlap

Performance restrictions:
\( x, x, b, refl \) - aligned on 8-byte boundary

Code/Data footprint: 0x1ed (.text)

Cycle count example:
Invocation parameters: \( N: 80; M: 6; x, d, b, refl \) - aligned on a 8-bytes boundary
Cycles Count: 3731

2.3 Vector mathematics

2.3.1 Vector dot product

Description
This routine takes two vectors and calculates their dot product.

Algorithm
\[
\begin{align*}
    r &= \sum_{n=0}^{N-1} x_n y_n \\
    &= \sum_{n=0}^{N-1} x_n y_n
\end{align*}
\]

Prototype

```c
f24 vec_dot ( const f24 * restrict x, const f24 * restrict y, int N )
```
2.3.2 Vector sum

Description This routine makes pair wise summation of vectors.

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

Prototype

```
void vec_add (      f24 * restrict z,
const f24 * restrict x,
const f24 * restrict y,
int N)
```

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>x</td>
<td>N</td>
<td>input data</td>
</tr>
<tr>
<td>f24</td>
<td>y</td>
<td>N</td>
<td>input data</td>
</tr>
<tr>
<td>int</td>
<td>N</td>
<td></td>
<td>length of vectors</td>
</tr>
</tbody>
</table>

Returned value:
Dot product of all data pairs, $Q_{30}$

Restrictions:
$x, y$ - aligned on 8-byte boundary
$N$ - divisible by 2

Performance restrictions:
$x, y$ - aligned on 8-byte boundary
$N$ - divisible by 2

Cycle count example:
Invocation parameters: $N$: 200
Cycles: 238

2.3.3 Power of a vector

Description This routine computes power of vector.

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

Prototype

```
f24     vec_power ( const f24 * restrict x, int N)
```

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>x</td>
<td>N</td>
<td>input data, $Q_{31}$</td>
</tr>
<tr>
<td>int</td>
<td>N</td>
<td></td>
<td>length of vector</td>
</tr>
</tbody>
</table>

Returned value:
Sum of squares of a vector, $Q_{30}$

Restrictions
$x$ - aligned on 8-byte boundary
$N$ - divisible by 2

Performance restrictions:
$x$ - aligned on 8-byte boundary
$N$ - divisible by 2

Code/Data footprint:
$0x7d$ (.text)
2.3.4 Vector scaling with saturation

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

**Algorithm**
\[ r_n = x_n \cdot 2^t \]

**Prototype**
```c
void vec_shift ( f24 * restrict y, const f24 * restrict x, int t, int N);
```

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>f24</td>
<td>x</td>
<td>N</td>
<td>input data</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>t</td>
<td></td>
<td>shift count. If positive, it shifts left with saturation, if negative it shifts right</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>N</td>
<td></td>
<td>length of vector</td>
</tr>
<tr>
<td>Output:</td>
<td>f24</td>
<td>y</td>
<td>N</td>
<td>output data</td>
</tr>
</tbody>
</table>

**Returned value**
None

**Restrictions**
- \(x,y\) should not overlap
- \(x,y\) - aligned on 8-byte boundary
- \(N\) - divisible by 2

**Code/Data footprint:**
0xb0 (.text)

**Cycle count example:**
Invocation parameters: N: 200  
Cycles: 138

---

2.3.5 Reciprocal on Q31 numbers

**Description**
This routine returns the fractional and exponential portion of the reciprocal of an vector \(x\) of Q31 numbers. Since the reciprocal is always greater than 1, it returns fractional portion \(frac\) in \(Q(31-\text{exp})\) format and exponent \(\text{exp}\) so true reciprocal value in the Q0.31 may be found by shifting fractional part left by exponent value.

For a reciprocal of 0, the result is not defined (but still be close to the maximum representative number)

**Example:**
\[ x=-1847296, \text{i.e.} -0.0008602142333984375Q31 \]
true reciprocal
\[ -1162.50111 \]
results:
\[ \text{frac}=-1218970880 \]
\[ \text{exp}=11 \]
so calculated reciprocal is
\[ -1218970880/2^{11-11}=-1162.50122 \]
relative error is \(9.5\cdot10^{-8}\)

**Worst case accuracy is 10LSB \((5\cdot10^{-7})\) of fractional part**

**Algorithm**
\[ \text{frac}_n \cdot 2^{\exp_n} = 1/x_n, n=0...N-1 \]

**Prototype**
```c
void vec_recip24 ( f24 * restrict frac, short *exp, const f24 * restrict x, int N);
```

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>f24_t</td>
<td>x</td>
<td>N</td>
<td>input data, Q31</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>N</td>
<td></td>
<td>length of vectors</td>
</tr>
<tr>
<td>Output:</td>
<td>f24</td>
<td>frac</td>
<td>N</td>
<td>fractional part of result, Q(31-\text{exp})</td>
</tr>
<tr>
<td></td>
<td>short</td>
<td>exp</td>
<td>N</td>
<td>exponent of result</td>
</tr>
</tbody>
</table>
2.3.6 Division of Q31 numbers

Description
This routine performs pair wise division of vectors written in Q31 format. It returns the fractional and exponential portion of the division result. Since the division may generate result greater than 1, it returns fractional portion \( \text{frac} \) in \( Q(31-\text{exp}) \) format and exponent \( \text{exp} \) 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.

Example:
\[
x = -1847296, \\
y = 261957888 \\
\text{true ratio} \quad \frac{x}{y} = -0.0070518814027309417724609375 \\
\text{results:} \\
\text{frac} = -969203200 \\
\text{exp} = -6 \\
\text{so calculated reciprocal is} \\
-969203200 / 2^{31-6} = -0.0070518814027309417724609375 \\
\text{relative error is} \quad 2.2 \times 10^{-8}
\]

Algorithm
Worst case accuracy is 10LSB (5\times 10^{-7}) of fractional part

Prototype
void vec_divide (f24 * restrict frac, 
short *exp, 
const f24 * restrict x, 
const f24 * restrict y, 
int N);

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>x</td>
<td>N</td>
<td>nominator</td>
</tr>
<tr>
<td>f24</td>
<td>y</td>
<td>N</td>
<td>denominator</td>
</tr>
<tr>
<td>int</td>
<td>N</td>
<td></td>
<td>length of vectors</td>
</tr>
<tr>
<td>f24</td>
<td>frac</td>
<td>N</td>
<td>fractional parts of result, ( Q(31-\text{exp}) )</td>
</tr>
<tr>
<td>short</td>
<td>exp</td>
<td>N</td>
<td>exponents of result</td>
</tr>
</tbody>
</table>

Returned value
None

Restrictions
\( x, \text{frac}, \text{exp} \) should not overlap

Performance restrictions:
\( \text{frac}, x \) - aligned on 8-byte boundary
\( N \) - divisible by 2 and \( \geq 2 \)
2.3.7 Logarithm

Description
Different kinds of logarithm (base 2, natural, base 10). Results are represented in
Q9 format or 0x8000 is returned on negative of zero argument.
This means, particularly, that
\[
\log_2(256) \rightarrow 0
\]
\[
\log_2(512) \rightarrow 0x2000000*(9-8)
\]
\[
\log_2(2^{30}) \rightarrow 0x2000000*(30-8)
\]

Accuracy 4.5e-5 worst case.

Algorithm
\[
z_n = \log_K x_n, n = 0...N-1, K = 2, e, 10
\]

Prototypes

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void vec_log2</td>
<td>const f24 * restrict x,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>int N</td>
<td></td>
</tr>
<tr>
<td>void vec_logn</td>
<td>const f24 * restrict x,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>int N</td>
<td></td>
</tr>
<tr>
<td>void vec_log10</td>
<td>const f24 * restrict x,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>int N</td>
<td></td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>x</td>
<td>N</td>
<td>input data, Q8</td>
</tr>
<tr>
<td>int</td>
<td>N</td>
<td></td>
<td>length of vectors</td>
</tr>
</tbody>
</table>

Returned value
none

Restrictions

<table>
<thead>
<tr>
<th>x, y</th>
<th>aligned on 8-byte boundary</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>divisible by 2</td>
</tr>
</tbody>
</table>

Performance restrictions:

<table>
<thead>
<tr>
<th>Code/Data footprint:</th>
<th>vec_logn(): 0x102(.rodata) + 0x486(.text)</th>
<th>N: 200</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec_log2():</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vec_log10():</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cycle count example:

<table>
<thead>
<tr>
<th>Code/Data footprint:</th>
<th>vec_logn(): 2530 (12.7 cycles/pts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec_log2():</td>
<td>2531 (12.7 cycles/pts)</td>
</tr>
<tr>
<td>vec_log10():</td>
<td>2528 (12.7 cycles/pts)</td>
</tr>
</tbody>
</table>

Scalar versions

<table>
<thead>
<tr>
<th>Prototypes</th>
<th>f24 scl_log2(f24 x)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>f24 scl_log(n f24 x)</td>
</tr>
<tr>
<td></td>
<td>f24 scl_log10(f24 x)</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>x</td>
<td>input data, Q8</td>
</tr>
</tbody>
</table>

Returned value
result, Q25
Performance restrictions: none
Code/Data footprint:
  scl_logn() :0x14 (.text)
  scl_log2() :0x14 (.text)
  scl_log10() :0x14 (.text)
Cycle count example:
  scl_logn() : 35
  scl_log2() : 35
  scl_log10() : 35

2.3.8 Sine/cosine
Description Calculates $\sin(\pi x)$ or $\cos(\pi x)$ for number written in Q31 format. Returns result in Q31 format as well.
Accuracy $6.1 \times 10^{-5}$ worst case.
Algorithm
\[ z_n = \sin(\pi x_n), n = 0 \ldots N-1 \] or
\[ z_n = \cos(\pi x_n), n = 0 \ldots N-1 \]
Prototypes
void vec_sine ( f24 * restrict y, const f24 * restrict x, int N)
void vec_cosine (f24 * restrict y, const f24 * restrict x, int N)
Arguments
\begin{tabular}{|c|c|c|c|}
\hline
Type & Name & Size & Description \\
\hline
input: & f24 & x & N \quad input \ data, \ Q31 \\
 & int & N & length of vectors \\
\hline
output: & f24 & y & N \quad Result, \ Q31 \\
\hline
\end{tabular}
Returned value None
Restrictions
  \( x, y \) should not overlap
Performance restrictions:
  \( x, y \) - aligned on 8-byte boundary
  \( N \) - divisible by 2
Code/Data footprint:
vec_sine() : 0x404 (.rodata) + 0x254 (.text)
vec_cosine() : 0x404 (.rodata) + 0x254 (.text)
Cycle count example:
Invocation parameters: N: 64; x,y aligned on a 8-bytes boundary
vec_sine() : 474 (7.4 cycles/pts)
vec_cosine() : 477 (7.4 cycles/pts)
Scalar versions
Prototypes
f24 scl_sine (f24 x)
f24 scl_cosine(f24 x)
Arguments
\begin{tabular}{|c|c|c|}
\hline
Type & Name & Description \\
\hline
input: & f24 & x \quad input \ data, \ Q31 \\
\hline
\end{tabular}

2.3.9 Tangent
Description Calculates $\tan(\pi x)$ for number written in Q31 format. Returns separately matissa and exponent values of result.
Absolute accuracy $1.9 \times 10^{-5}$ for \( x = \{ -n/4: n/4 \}, \{ -n5/4: n\times7/4 \} 
Relative $5 \times 10^{-3}$ for other values
Algorithm
\[ z_n = \tan(\pi x_n), n = 0 \ldots N-1 \]
Prototype
void vec_tan (f24 * restrict y, short * restrict exp, const f24 * restrict x, int N)
Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>N</td>
<td></td>
<td>length of vectors</td>
</tr>
</tbody>
</table>

Output:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>x</td>
<td>N</td>
<td>input data, Q31</td>
</tr>
<tr>
<td>short</td>
<td>exp</td>
<td>N</td>
<td>exponent of result</td>
</tr>
</tbody>
</table>

Returned value none

Restrictions

x, y should not overlap

Performance restrictions:

N - divisible by 2

Code/Data footprint:

0x208 (.rodata) + 0x358 (.text)

Cycle count example:

Parameters invocation: N: 64
Cycles count: 4251 (66 cycles/pts)

Scalar versions

Prototype

int32_t scl_tan (int32_t x)

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32_t</td>
<td>x</td>
<td>input data</td>
</tr>
</tbody>
</table>

Returned value packed result, exp resides in 8 most significant bits

Code/Data footprint:

0x32e (.text)

Cycle count example:

125

2.3.10 Full arctangent

Description

Function calculates four quadrant arctangent of complex numbers \( \frac{\text{atan2}(x)}{\pi} \). Returns result in Q31 format.

Absolute phase accuracy: \( 1.8 \times 10^{-6} \)

If both arguments are zero it returns zero as well.

Algorithm

\[ z_\pi = \arg(x_\pi) / \pi, n = 0...N - 1 \]

Prototype

void vec_atan2 (f24 * restrict z, const f24 * restrict x, int N)

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>x</td>
<td>2*N</td>
<td>complex input data. Real and imaginary parts are interleaved and real part goes first.</td>
</tr>
<tr>
<td>int</td>
<td>N</td>
<td></td>
<td>length of vectors</td>
</tr>
</tbody>
</table>

Output:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>z</td>
<td>N</td>
<td>result</td>
</tr>
</tbody>
</table>

Returned value none

Restrictions

x, y should not overlap

Performance restrictions:

N - aligned on 8-byte boundary

Code/Data footprint:

0x580

Cycle count example:

Invocation parameters: N: 64
Cycles count: 5521 (86 cycles/pts)

Scalar versions

Prototype

f24 scl_atan2 (f24 re, f24 im)

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 2.3.11 Arctangent

**Description**
Function calculates arctangent of number written in Q31 format. It scales output to $\pi$ so it is always in range $-0x20000000 \ldots 0x20000000$ which corresponds to the real phases $\pm \pi/4$

Absolute phase accuracy $1.810^{-6}$
This function is much faster than `vec_atan2()`, so it is preferable when the full phase is not required.

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

**Prototype**
```c
void vec_atan (f24 * restrict z, const f24 * restrict x, int N )
```

**Arguments**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f24</td>
<td>x</td>
<td>N</td>
<td>input data</td>
</tr>
<tr>
<td>Int</td>
<td>N</td>
<td></td>
<td>length of vectors</td>
</tr>
<tr>
<td>Output:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f24</td>
<td>z</td>
<td>N</td>
<td>result</td>
</tr>
</tbody>
</table>

**Returned value**
None

**Restrictions**
$x,y$ should not overlap

**Performance restrictions:**
$x,y$ - aligned on 8-byte boundary
$N$ - divisible by 2

**Code/Data footprint:**
0x230(.rodata) + 0x580(.text)

**Cycle count example:**
Invocation parameters: $N$: 64
539 (8.4 cycles/pts)

**Scalar versions**

**Prototype**
```c
f24 scl_atan (f24 x)
```

**Arguments**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f24</td>
<td>x</td>
<td>input data</td>
</tr>
</tbody>
</table>

**Returned value**
result

**Code/data footprint:**
0x5a6 (.text)

**Cycle count example:**
26

### 2.3.12 Common exponent

**Description**
This function determines the number of redundant sign bits for each value (as if it was loaded in a 32-bit register) and returns the minimum number over the whole vector. This may be useful for a FFT implementation to normalize data.

**Algorithm**
$$ z_n = \min_{n=0 \ldots N-1} (\text{norm}(x_n)) $$

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

**Prototype**
```c
int vec_bexp (const int16_t * restrict x, int N )
```

**Arguments**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f24</td>
<td>x</td>
<td>N</td>
<td>input data, Q31</td>
</tr>
</tbody>
</table>
2.4.1 Matrix multiply

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

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

Prototype:
```
void mtx_mpy (       f24* restrict z,
                  const f24 *restrict x,
                  const f24 *restrict y,
                  int N, int M, int P );
```

Arguments:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f24</td>
<td>x</td>
<td>N*M</td>
<td>input matrix, Q31</td>
</tr>
<tr>
<td>f24</td>
<td>y</td>
<td>M*P</td>
<td>input matrix, Q31</td>
</tr>
<tr>
<td>int</td>
<td>N</td>
<td></td>
<td>number of rows in matrix $x$ and $z$</td>
</tr>
<tr>
<td>int</td>
<td>M</td>
<td></td>
<td>number of columns in matrix $x$ and number of rows in matrix $y$</td>
</tr>
<tr>
<td>int</td>
<td>P</td>
<td></td>
<td>number of columns in matrices $y$ and $z$</td>
</tr>
</tbody>
</table>

Output:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32</td>
<td>z</td>
<td>N*P</td>
<td>output matrix, Q30</td>
</tr>
</tbody>
</table>

Returned value
none

Restrictions
$x, y, z$ should not overlap
$x, y$ - aligned on 8-byte boundary
$N, M, P$ - divisible by 2 and $\geq 4$

Performance restrictions:

Code/data footprint:
0x5f2 (.text)

Cycle count example:
Invocation parameters: N: 6; M: 10; P: 8; x, y aligned on a 8-bytes boundary
Cycles Count: 964
2.4.2 Matrix transpose

Description
This function transposes matrix.

Algorithm
\[ y_{m,n} = x_{n,m}, n = 0...N-1, m = 0...M-1 \]

Prototype

```c
void mtx_trans ( f24 restrict y, const f24 x, int N, int M );
```

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>f24</td>
<td>N*M</td>
<td>input matrix</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>N</td>
<td>number of rows in vector ( x ) and number of columns in vector ( y )</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>M</td>
<td>number of columns in vector ( x ) and number of rows in vector ( y )</td>
</tr>
</tbody>
</table>

Output:

| f24    | y      | M*N       | output matrix                                                               |

Returned value
none

Restrictions
\( x, y \) should not overlap

Performance restrictions:
The function is most efficient if \( M, N \) are multiples of two and the input array \((N>\)M\)) or the output array \((N<=M)\) is aligned on a 8-bytes boundary.

Code/Data footprint:
0x26d (.text)

Cycle count example:
Invocation parameters: N: 30; M: 40; \( x, y \) aligned on a 8-bytes boundary
Cycles Count: 1960

2.5 Fast Fourier Transforms

2.5.1 FFT on complex data

Description
This function makes FFT on complex data.

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

Algorithm
\[ y = FFT(x) \]

Prototype

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

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>int16_t</td>
<td>2*N</td>
<td>complex input signal. Real and imaginary data are interleaved and real data goes first</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td></td>
<td>FFT size</td>
</tr>
<tr>
<td>Output:</td>
<td>int16_t</td>
<td>2*N</td>
<td>output spectrum. Real and imaginary data are interleaved and real data goes first</td>
</tr>
</tbody>
</table>

Returned value
none

Restrictions
\( x, y \) should not overlap
\( x, y \) - aligned on a 8-bytes boundary
\( N \) - 16, 32, 64, 128, 256 or 512.

Performance restrictions:
none

Code/data
0x570e (.rodata) + 0xb0f (.text)
2.5.2 FFT on real data

**Description**
This function makes FFT on real data forming half of spectrum

**NOTES:**
1. Bit-reversing reordering is done here.
2. FFT does not make scaling of input data and it should be done externally to avoid possible overflows.
3. FFT runs in-place algorithm so **INPUT DATA WILL APPEAR DAMAGED** after the call.
4. Real data FFT function calls `fft_cplx()` to apply complex FFT of size \( N/2 \) to input data and then transforms the resulting spectrum.

**Algorithm**
\[ y = \text{FFT}(\text{real}(x)) \]

**Prototype**
```c
void fft_real(f24* y, f24* x, int N);
```

**Arguments**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>f24</td>
<td>x</td>
<td>N</td>
</tr>
<tr>
<td>int</td>
<td>N</td>
<td>FFT size</td>
<td></td>
</tr>
<tr>
<td>Output:</td>
<td>f24</td>
<td>y</td>
<td>(N/2+1)*2</td>
</tr>
</tbody>
</table>

**Returned value:** none

**Restrictions:**
\( x, y \) should not overlap
\( x, y \) - aligned on 8-bytes boundary
\( N \) - 32, 64, 128, 256, 512 or 1024.

**Performance restrictions:** None

**Code/Data footprint:** 0x2A6 (.text)

**Cycle count:**

<table>
<thead>
<tr>
<th>N</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>N=512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>11118</td>
<td></td>
</tr>
</tbody>
</table>

2.5.3 Inverse FFT on complex data

**Description**
This function makes inverse FFT on complex data.

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

**Algorithm**
\[ y = \text{FFT}^{-1}(x) \]

**Prototype**
```c
void ifft_cplx(f24* y, f24* x, int N);
```

**Arguments**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>f24</td>
<td>x</td>
<td>2*N</td>
</tr>
</tbody>
</table>

**Cycle count:**

<table>
<thead>
<tr>
<th>N</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>N=512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6206</td>
<td></td>
</tr>
</tbody>
</table>
2.5.4 Inverse FFT forming real data

Description
This function makes inverse FFT on half spectral data forming real data samples.

NOTES:
1. Bit-reversing reordering is done here.
2. IFFT does not make scaling of input data and it should be done externally to avoid possible overflows.
3. IFFT runs in-place algorithm so INPUT DATA WILL APPEAR DAMAGED after call.
4. Inverse FFT function for real signal transforms the input spectrum and then calls ifft_cplx() with FFT size set to N/2.

Algorithm
\[ y = \text{real}(FFT^{-1}(x)) \]

Prototype
```c
void ifft_real(f24* y, f24* x, int N);
```

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>f24</td>
<td>x</td>
<td>(N/2+1)*2 input spectrum. Real and imaginary data are interleaved and real data goes first</td>
</tr>
<tr>
<td>Int</td>
<td>N</td>
<td></td>
<td>FFT size</td>
</tr>
<tr>
<td>Output:</td>
<td>f24</td>
<td>y</td>
<td>N real output signal</td>
</tr>
</tbody>
</table>

Returned value
none

Restrictions
\( x, y \) should not overlap
\( x, y \) aligned on 8-bytes boundary
\( N = 32, 64, 128, 256, 512 \) or 1024.

Performance restrictions:
None

Code/data footprint:
0x172 (.text)

Cycle count:

<table>
<thead>
<tr>
<th>N</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>N=512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 3.1 Functions performance list

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Code/Data Footprint</th>
<th>Cycles Measurements</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FIR filters and related functions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fir_bk</td>
<td>0x273 (.text)</td>
<td>12437</td>
</tr>
<tr>
<td>fir_cbk</td>
<td>0x237 (.text)</td>
<td>22013</td>
</tr>
<tr>
<td>fir_sr</td>
<td>0x5de (.text)</td>
<td>13433</td>
</tr>
<tr>
<td>fir_dec</td>
<td>0x4ad (.text)</td>
<td>11196</td>
</tr>
<tr>
<td>fir_interp</td>
<td>0x27f (.text)</td>
<td>20090</td>
</tr>
<tr>
<td>fir_ssc</td>
<td>0x19d (.text)</td>
<td>443</td>
</tr>
<tr>
<td><strong>IIR filters</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iir_bqc</td>
<td>0x4d2 (.text)</td>
<td>7450</td>
</tr>
<tr>
<td>iir_bqd</td>
<td>0x2eb (.text)</td>
<td>3412</td>
</tr>
<tr>
<td>iir_latr</td>
<td>0xb97 (.text)</td>
<td>2066</td>
</tr>
<tr>
<td>iir_latc</td>
<td>0x1077 (.text)</td>
<td>3764</td>
</tr>
<tr>
<td>iir_latd</td>
<td>0x1ed (.text)</td>
<td>3731</td>
</tr>
<tr>
<td><strong>Vector mathematics</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vec dot</td>
<td>0xcdf (.text)</td>
<td>238</td>
</tr>
<tr>
<td>vec add</td>
<td>0x7d (.text)</td>
<td>333</td>
</tr>
<tr>
<td>vec_power</td>
<td>0x65 (.text)</td>
<td>138</td>
</tr>
<tr>
<td>vec_shift</td>
<td>0xb0 (.text)</td>
<td>238</td>
</tr>
<tr>
<td>vec recip24</td>
<td>0x2b5 (.text)</td>
<td>296 (14.8 cycles/pts)</td>
</tr>
<tr>
<td>vec divide</td>
<td>0x2c0 (.text)</td>
<td>4882 (24.4 cycles/pts)</td>
</tr>
<tr>
<td>vec logn</td>
<td>0x102 (.rodata) + 0x486 (.text)</td>
<td>2530 (12.7 cycles/pts)</td>
</tr>
<tr>
<td>vec log2</td>
<td>0x486 (.text)</td>
<td>2531 (12.7 cycles/pts)</td>
</tr>
<tr>
<td><strong>Matrix operations</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtx mpy</td>
<td>0x5f2 (.text)</td>
<td>964</td>
</tr>
<tr>
<td>mtx trans</td>
<td>0x26d (.text)</td>
<td>1960</td>
</tr>
<tr>
<td><strong>FFT</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fft_cplx</td>
<td>0x570a (.rodata) + 0xb0f (.text)</td>
<td>11118</td>
</tr>
<tr>
<td>Function Name</td>
<td>Code/Data Footprint</td>
<td>Invocation parameters</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>fft_real</td>
<td>0x2A6 (.text)</td>
<td>N=512</td>
</tr>
<tr>
<td>ifft_cplx</td>
<td>0x570e(.rodata) + 0xb0f(.text)</td>
<td>N=512</td>
</tr>
<tr>
<td>ifft_real</td>
<td>0x172 (.text)</td>
<td>N=512</td>
</tr>
</tbody>
</table>
Chapter 4  Test environment and examples

4.1 How to build and run tests

It is assumed that you have downloaded and unpacked the NatureDSP Signal library delivery. For your convenience, the `<ROOT_DIR>` stands hereafter for the root directory of the delivery. It is also assumed that Xtensa building tools with support for the HiFi2 core are installed and properly configured as specified by appropriate documents from Tensilica.

Compilation and run possible for both Windows and Linux operating systems and for both little and big endian memory model. So we have 4 possible combinations:

<table>
<thead>
<tr>
<th>OS</th>
<th>Little</th>
<th>Endianess</th>
<th>Big</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>hifi2</td>
<td>hifi2be</td>
<td>hifi2be</td>
</tr>
<tr>
<td>Linux</td>
<td>hifi2_linux</td>
<td>hifi2be_linux</td>
<td></td>
</tr>
</tbody>
</table>

Name of this configuration denoted as `<CFG>` determines the name of directories for makefiles and tests.

All makefiles use build tools 2010.1. If you use newer one change the lines

`XTENSABUILD=RC-2010.1-win32` for Windows makefile and

`XTENSABUILD=RC-2010.1-linux` for Linux environment

4.1.1 Building the NatureDSP Signal library

In order to build the object library for Tensilica HiFi2 core, perform the following steps:

1. Switch to the library make file location:
   `<ROOT_DIR>/src/NatureDSP_Signal/NatureDSP_Signal_<CFG>/makefile`

2. Open the makefile for editing and locate the following line:
   `XTENSA_INSTALL = c:/xtensa/XtDevTools/install`
   Adjust the path according to your installation of Xtensa tools and save the make file.

3. Though the library make file defines a few variants of building actions and options sets, you can just issue a `make` command with no arguments to launch the building process. Please refer to the list of available options described in the comments block of the make file.

4. After the make utility finishes, switch to the object library location to verify that it is up-to-date:
   `<ROOT_DIR>/src/Libs/NatureDSP_Signal_<CFG>.a`

4.1.2 Building and running tests

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

1. Switch to the NatureDSP Signal object library file location and verify that it is up-to-date, otherwise refer to 4.1.1 for instruction on how to build it:
   `<ROOT_DIR>/src/Libs/NatureDSP_Signal_<CFG>.a`

2. Switch to the library tests make file location:
   `<ROOT_DIR>/src/test_<CFG>/makefile`

3. Open the makefile for editing and locate the following line:
   `XTENSA_INSTALL = c:/xtensa/XtDevTools/install`
Adjust the path according to your installation of Xtensa tools and save the make file.

4. Issue a **make test** command to build the test executable and immediately run it using the instruction set simulator. Please note that the full test may take more than an hour before it finishes. Normally you should observe the list of completed tests and the name of the test in progress during this time.

5. After all the tests have been run and the program terminates you can find out the cycle count for a particular function from a text file containing the profiling results: `<ROOT_DIR>/src/test_xd2/test_<CFG>.txt`
4.2 FIR tests

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

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

4.3 Matrix operations tests

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

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Test procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_mtx_mpy</td>
<td>Tests function mtx_mpy(). Performs matrix multiplication (5x10 to 10x6 forming 5x6 result) and compares results with precalculated data.</td>
</tr>
</tbody>
</table>
4.4 Vector operations tests

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

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

4.5 IIR tests

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

4.6 FFT tests

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Test procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_fft_cplx</td>
<td>Tests functions <code>fft_cplx()</code>, <code>ifft_cplx()</code>. Passes the signal read from file by blocks through the FFT filter and write filtered output to the file.</td>
</tr>
<tr>
<td>test_fft_real</td>
<td>Tests functions <code>fft_real()</code> and <code>ifft_real()</code>. Passes the signal read from file by blocks through the FFT filter and write filtered output to the file.</td>
</tr>
</tbody>
</table>
Chapter 5  Customer support

If you have questions, want to report problems or suggestions regarding the NatureDSP Signal library or want to port this library to another platforms, contact IntegrIT Ltd. at support@integrit.ru. Visit www.integrit.com to get more information about products and services.