xCORE-200 DSP Library

This API reference manual describes the XMOS fixed-point digital signal processing software library. The library implements of a suite of common signal processing functions for use on XMOS xCORE-200 multi-core microcontrollers.

Required tools and libraries

- xTIMEcomposer Tools Version 14.0.1 or later

Required hardware

Only XMOS xCORE-200 based multicore microcontrollers are supported with this library. The previous generation XS1 based multicore microntrollers are not supported.

The xCORE-200 has a single cycle 32x32->64 bit multiply/accumulate unit, single cycle double-word load and store, dual issue instruction execution, and other instruction set enhancements. These features make xCORE-200 an efficient platform for executing digital signal processing algorithms.

Prerequisites

This document assumes familiarity with the XMOS xCORE architecture, the XMOS tool chain, the 'C' programming language, and digital signal processing concepts.

Software version and dependencies

This document pertains to version 1.0.3 of this library. It is known to work on version 14.1.1 of the xTIMEcomposer tools suite, it may work on other versions.

The library does not have any dependencies (i.e. it does not rely on any other libraries).

Related application notes

The following application notes use this library:

- AN00209 - xCORE-200 DSP Library

1 Overview

1.1 Introduction

This API reference manual describes the XMOS xCORE-200 fixed-point digital signal processing firmware library. The library implements of a suite of common signal processing functions for use on XMOS xCORE-200 multicore microcontrollers.

1.2 Library Organization

The library is divided into function collections with each collection covering a specific digital signal processing algorithm category. The API and implementation for each category are provided by a single 'C' header file and implementation file.

Category	Source Files	Functions
Fixed point	lib_dsp_qformat	Q16 through Q31 formats, fixed and floating point conversions
Filters	lib_dsp_filters	FIR, biquad, cascaded biquad, and convolution
Adaptive	lib_dsp_adaptive	LMS and NLMS Adaptive filters
Scalar math	lib_dsp_math	Multiply, square root, reciprocal, inverse square root
Vector math	lib_dsp_vector	Scalar/vector add/subtract/multiply, dot product
Matrix math	lib_dsp_matrix	Scalar/matrix add/subtract/multiply, inverse and transpose
Statistics	lib_dsp_statistics	Vector mean, sum-of-squares, root-mean-square, variance
Design	lib_dsp_design	Biquad coefficient generation for various filter types

Table 1: DSP library organization :class: narrow

2 Fixed-Point Format

2.1 Q Format Introduction

The library functions support 32 bit input and output data, with internal 64 bit accumulator. The output data can be scaled to any of the supported Q Formats (Q16 through Q31). Further details about Q Format numbers is available here : https://en.wikipedia.org/wiki/Q_(number_format).

2.2 The 'q_format' Parameter

All XMOS DSP library functions that incorporate a multiply operation accept a parameter called q_format. This parameter can naively be used to specify the fixed point format for all operands and results (if applicable) where the formats are the same for all parameters. For example:

```
result_q28 = lib_dsp_math_multiply( input1_q28, input2_q28, 28 );
```

The 'q_format' parameter, being used after one or more sequences of multiply and/or multiply-accumulate, is used to right-shift the 64-bit accumulator before truncating the value back to a 32-bit integer (i.e. the 32-bit fixed-point result). Therefore the 'q_format' parameter can be used to perform the proper fixed-point adjustment for any combination of input operand fixed-point format and desired fixed-point result format.

The output fixed-point fraction bit count is equal to the sum of the two input fraction bit counts minus the desired result fraction bit count:

```
q_format = input1 fraction bit count +  input2 fraction bit count - result fraction bit count
```

For example:

```
// q_format_parameter = 31 = 30 + 29 - 28
result_q28 = lib_dsp_math_multiply( input1_q30, input2_q29, 31 );

// q_format_parameter = 27 = 28 + 29 - 30
result_q30 = lib_dsp_math_multiply( input1_q28, input2_q29, 27 );
```

3 Filter Functions: Finite Impulse Response (FIR) Filter

Function	lib_dsp_filters_fir

Description	This function implements a Finite Impulse Response (FIR) filter. The function operates on a single sample of input and output data (i.e. each call to the function processes one sample). The FIR filter algorithm is based upon a sequence of multiply-accumulate (MAC) operations. Each filter coefficient h[i] is multiplied by a state variable which equals a previous input sample x[i], or y[n]=x[n]*h[0]+x[n-1]*h[1]+x[n-2]*h[2]+x[n-N+1]*h[N-1] The parameter `filter_coeffs` points to a coefficient array of size N = num_taps. The filter coefficients are stored in forward order (e.g. h[0],h[1],h[N-1]). The following example shows a five-tap (4th order) FIR filter with samples and coefficients represented in Q28 fixed-point format.
	
	`int filter_coeff[5] = { Q28(0.5),Q(-0.5),Q28(0.0),Q28(-0.5),Q28(0.5) };`
	`int filter_state[4] = { 0, 0, 0, 0 };`
	`int result = lib_dsp_fir( sample, filter_coeff, filter_state, 5, 28 );`
	
	The FIR algorithm involves multiplication between 32-bit filter coefficients and 32-bit state data producing a 64-bit result for each coeffient and state data pair. Multiplication results are accumulated in 64-bit accumulator with the final result shifted to the required fixed-point format. Therefore overflow behavior of the 32-bit multiply operation and truncation behavior from final shifing of the accumulated multiplication results must be considered.

Type	`int`
	`lib_dsp_filters_fir(int input_sample,`
	`                     const int filter_coeffs[],`
	`                     int state_data[],`
	`                     int tap_count,`
	`                     int q_format)`

Parameters	`input_sample`
	          The new sample to be processed.
	
	`filter_coeffs`
	          Pointer to FIR coefficients array arranged as [b0,b1,b2,bN-1].
	
	`state_data`
	          Pointer to filter state data array of length N. Must be initialized at startup to all zeros.
	
	`tap_count` Filter tap count (N = tap_count = filter order + 1).
	
	`q_format` Fixed point format (i.e. number of fractional bits).

Returns	The resulting filter output sample.

4 Filter Functions: Interpolating FIR Filter

Function	lib_dsp_filters_interpolate
Description	This function implements an interpolating FIR filter. The function operates on a single input sample and outputs a set of samples representing the interpolated data, whose sample count is equal to `interp_factor`. (i.e. and each call to the function processes one sample and results in `interp_factor` output samples). The FIR filter algorithm is based upon a sequence of multiply-accumulate (MAC) operations. Each filter coefficient h[i] is multiplied by a state variable which equals a previous input sample x[i], or y[n]=x[n]*h[0]+x[n-1]*h[1]+x[n-2]*h[2]+x[n-N+1]*h[N-1] `filter_coeffs` points to a coefficient array of size N = num_taps. The filter coefficients are stored in forward order (e.g. h[0],h[1],h[N-1]). The FIR algorithm involves multiplication between 32-bit filter coefficients and 32-bit state data producing a 64-bit result for each coefficient and state data pair. Multiplication results are accumulated in 64-bit accumulater with the final result shifted to the required fixed-point format. Therefore overflow behavior of the 32-bit multiply operation and truncation behavior from final shifing of the accumulated multiplication results must be considered.
Type	```void lib_dsp_filters_interpolate(int input_sample, const int filter_coeffs[], int state_data[], int tap_count, int interp_factor, int output_samples[], int q_format)```

Continued on next page

Parameters	input_sample
	The new sample to be processed.
	filter_coeffs
	Pointer to FIR coefficients array arranged as: hM,h(1L+M),h(2L+M),h((N-1)L+M), h1,h(1L+1),h(2L+1),h((N-1)L+1), h0,h(1L+0),h(2L+0),h((N-1)L+0), where M = N-1
	state_data
	Pointer to filter state data array of length N. Must be initialized at startup to all zeros.
	tap_count — Filter tap count (N = tap_count = filter order + 1).
	interp_factor
	The interpolation factor/index (i.e. the up-sampling ratio). The interpolation factor/index can range from 2 to 16.
	output_samples
	The resulting interpolated samples.
	q_format — Fixed point format (i.e. number of fractional bits).

5 Filter Functions: Decimating FIR Filter

Function	lib_dsp_filters_decimate
Description	This function implements an decimating FIR filter. The function operates on a single set of input samples whose count is equal to the decimation factor. (i.e. and each call to the function processes `decim_factor` samples and results in one sample). The FIR filter algorithm is based upon a sequence of multiply-accumulate (MAC) operations. Each filter coefficient $h[i]$ is multiplied by a state variable which equals a previous input sample $x[i]$, or $y[n]=x[n]*h[0]+x[n-1]*h[1]+x[n-2]*h[2]+x[n-N+1]*h[N-1]$ `filter_coeffs` points to a coefficient array of size N = num_taps. The filter coefficients are stored in forward order (e.g. `h[0]`,`h[1]`,`h[N-1]`). The FIR algorithm involves multiplication between 32-bit filter coefficients and 32-bit state data producing a 64-bit result for each coefficient and state data pair. Multiplication results are accumulated in 64-bit accumulater with the final result shifted to the required fixed-point format. Therefore overflow behavior of the 32-bit multiply operation and truncation behavior from final shifing of the accumulated multiplication results must be considered.
Type	```int lib_dsp_filters_decimate(int input_samples[], const int filter_coeffs[], int state_data[], int tap_count, int decim_factor, int q_format)```

Parameters	`input_samples`
	        The new samples to be decimated.
	
	`filter_coeffs`
	        Pointer to FIR coefficients array arranged as `[b0,b1,b2,bN-1]`.
	
	`state_data`
	        Pointer to filter state data array of length N. Must be initialized at startup to all zeros.
	
	`tap_count` Filter tap count (N = tap_count = filter order + 1).
	
	`decim_factor`
	        The decimation factor/index (i.e. the down-sampling ratio).
	
	`q_format` Fixed point format (i.e. number of fractional bits).

Returns	The resulting decimated sample.

6 Filter Functions: Bi-Quadratic (BiQuad) IIR Filter

Function	lib_dsp_filters_biquad

Description	This function implements a second order IIR (direct form I).
	The function operates on a single sample of input and output data (i.e. and each call to the function processes one sample).
	The IIR filter algorithm is based upon a sequence of multiply-accumulate (MAC) operations. Each filter coefficient b[i] is multiplied by a state variable which equals a previous input sample x[i], or y[i]=x[n]*b[0]+x[n–1]*b[1]+x[n–2]*b2+x[n–1]*a[1]+x[n–2]*a[2]
	The filter coefficients are stored in forward order (e.g. b0,b1,b2,a1,a2).
	Example showing a single Biquad filter with samples and coefficients represented in Q28 fixed-point format:
	<pre>int filter_coeff[5] = { Q28(+0.5), Q(-0.1), Q28(-0.5), Q28(-0.1), Q28(0.1) };
	int filter_state[4] = { 0, 0, 0, 0 };
	int result = lib_dsp_biquad(sample, filter_coeff, filter_state, 28);</pre>
	The IIR algorithm involves multiplication between 32-bit filter coefficients and 32-bit state data producing a 64-bit result for each coefficient and state data pair. Multiplication results are accumulated in 64-bit accumulater with the final result shifted to the required fixed-point format. Therefore overflow behavior of the 32-bit multiply operation and truncation behavior from final shifing of the accumulated multiplication results must be considered.

Type	<pre>int
	lib_dsp_filters_biquad(int input_sample,
	    const int filter_coeffs[LIB_DSP_NUM_COEFFS_PER_BIQUAD],
	    int state_data[LIB_DSP_NUM_STATES_PER_BIQUAD],
	    int q_format)</pre>

Parameters	input_sample The new sample to be processed. filter_coeffs Pointer to biquad coefficients array arranged as [b0,b1,b2,a1,a2]. state_data Pointer to filter state data array (initialized at startup to zeros). The length of the state data array is 4. q_format Fixed point format (i.e. number of fractional bits).
Returns	The resulting filter output sample.

7   Filter Functions: Cascaded BiQuad Filter

Function	lib_dsp_filters_biquads

Description	This function implements a cascaded direct form I BiQuad filter. The function operates on a single sample of input and output data (i.e. and each call to the function processes one sample). The IIR filter algorithm is based upon a sequence of multiply-accumulate (MAC) operations. Each filter coefficient b[i] is multiplied by a state variable which equals a previous input sample x[i], or y[n]=x[n]*b[0]+x[n-1]*b[1]+x[n-2]*b2+x[n-1]*a[1]+x[n-2]*a[2] The filter coefficients are stored in forward order (e.g. section1:b0,b1,b2,a1,a2,sectionN:b0,b1,b2,a1,a2). Example showing a 4x cascaded Biquad filter with samples and coefficients represented in Q28 fixed-point format:
	
	`int filter_coeff[20] = { Q28(+0.5), Q(-0.1), Q28(-0.5), Q28(-0.1), Q28(0.1),`
	`                        Q28(+0.5), Q(-0.1), Q28(-0.5), Q28(-0.1), Q28(0.1),`
	`                        Q28(+0.5), Q(-0.1), Q28(-0.5), Q28(-0.1), Q28(0.1),`
	`                        Q28(+0.5), Q(-0.1), Q28(-0.5), Q28(-0.1), Q28(0.1) };`
	`int filter_state[16] = { 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0 };`
	`int result = lib_dsp_cascaded_biquad( sample, filter_coeff, filter_state, 4, 28 );`
	
	The IIR algorithm involves multiplication between 32-bit filter coefficients and 32-bit state data producing a 64-bit result for each coefficient and state data pair. Multiplication results are accumulated in 64-bit accumulater with the final result shifted to the required fixed-point format. Therefore overflow behavior of the 32-bit multiply operation and truncation behavior from final shifing of the accumulated multiplication results must be considered.

Type	`int`
	`lib_dsp_filters_biquads(int input_sample,`
	`                        const int filter_coeffs[],`
	`                        int state_data[],`
	`                        int num_sections,`
	`                        int q_format)`

Parameters	`input_sample`
	      The new sample to be processed.
	
	`filter_coeffs`
	      Pointer to biquad coefficients array for all BiQuad sections. Arranged as [section1:b0,b1,b2,a1,a2,...sectionN:b0,b1,b2,a1,a2].
	
	`state_data`
	      Pointer to filter state data array (initialized at startup to zeros). The length of the state data array is `num_sections` * 4.
	
	`num_sections`
	      Number of BiQuad sections.
	
	`q_format` Fixed point format (i.e. number of fractional bits).

Returns	The resulting filter output sample.

8 Adaptive Filter Functions: LMS Adaptive Filter

Function	lib_dsp_adaptive_lms

Description	This function implements a least-mean-squares adaptive FIR filter. LMS filters are a class of adaptive filters that adjust filter coefficients in order to create a transfer function that minimizes the error between the input and reference signals. FIR coefficients are adjusted on a per sample basis by an amount calculated from the given step size and the instantaneous error. The function operates on a single sample of input and output data (i.e. and each call to the function processes one sample and each call results in changes to the FIR coefficients). The general LMS algorithm, on a per sample basis, is to:
	
	1) Apply the transfer function: output = FIR(input)
	2) Compute the instantaneous error value: error = reference – output
	3) Compute current coefficient adjustment delta: delta = mu * error
	4) Adjust transfer function coefficients:
	   FIR_COEFFS[n] = FIR_COEFFS[n] + FIR_STATE[n] * delta
	
	Example of a 100-tap LMS filter with samples and coefficients represented in Q28 fixed-point format:
	
	`int filter_coeff[100] = { ... not shown for brevity };`
	`int filter_state[100] = { 0, 0, 0, 0, ... not shown for brevity };`
	
	`int output_sample = lib_dsp_adaptive_lms`
	`(`
	`  input_sample, reference_sample, &error_sample,`
	`  filter_coeff_array, filter_state_array, 100, Q28(0.01), 28`
	`);`
	
	The LMS filter algorithm involves multiplication between two 32-bit values and 64-bit accumulation as a result of using an FIR as well as coefficient step size calculations). Multiplication results are accumulated in 64-bit accumulater with the final result shifted to the required fixed-point format. Therefore overflow behavior of the 32-bit multiply operation and truncation behavior from final shifing of the accumulated multiplication results must be considered for both FIR operations as well as for coefficient step size calculation and FIR coefficient adjustment.

Type	`int`
	`lib_dsp_adaptive_lms(int input_sample,`
	`                     int reference_sample,`
	`                     int error_sample[],`
	`                     int filter_coeffs[],`
	`                     int state_data[],`
	`                     int tap_count,`
	`                     int step_size,`
	`                     int q_format)`

Continued on next page

Parameters	input_sample
	The new sample to be processed.
	reference_sample
	Reference sample.
	error_sample
	Pointer to resulting error sample (error = reference - output)
	filter_coeffs
	Pointer to FIR coefficients arranged as [b0,b1,b2, ...,bN-1].
	state_data
	Pointer to FIR filter state data array of length N. Must be initialized at startup to all zeros.
	tap_count Filter tap count where N = tap_count = filter order + 1.
	step_size Coefficient adjustment step size, controls rate of convergence.
	q_format Fixed point format (i.e. number of fractional bits).
Returns	The resulting filter output sample.

9 Adaptive Filter Functions: Normalized LMS Filter

Function	lib_dsp_adaptive_nlms

Description	This function implements a normalized LMS FIR filter.
	
	LMS filters are a class of adaptive filters that adjust filter coefficients in order to create the a transfer function that minimizes the error between the input and reference signals. FIR coefficients are adjusted on a per sample basis by an amount calculated from the given step size and the instantaneous error.
	
	The function operates on a single sample of input and output data (i.e. and each call to the function processes one sample and each call results in changes to the FIR coefficients).
	
	The general NLMS algorithm, on a per sample basis, is to:
	
	1) Apply the transfer function: output = FIR(input)
	2) Compute the instantaneous error value: error = reference – output
	3) Normalize the error using the instantaneous power computed by:
	   E = x[n]^2 + ... + x[n–N+1]^2
	4) Update error value: error = (reference – output) / E
	5) Compute current coefficient adjustment delta: delta = mu * error
	6) Adjust transfer function coefficients:
	   FIR_COEFFS[n] = FIR_COEFFS[n] + FIR_STATE[n] * delta
	
	Example of a 100-tap NLMS filter with samples and coefficients represented in Q28 fixed-point format:
	
	```int filter_coeff[100] = { ... not shown for brevity };```
	```int filter_state[100] = { 0, 0, 0, 0, ... not shown for brevity };```
	
	```int output_sample = lib_dsp_adaptive_nlms```
	```(```
	``` input_sample, reference_sample, &error_sample,```
	``` filter_coeff_array, filter_state_array, 100, Q28(0.01), 28```
	```);```
	
	The LMS filter algorithm involves multiplication between two 32-bit values and 64-bit accumulation as a result of using an FIR as well as coefficient step size calculations). Multiplication results are accumulated in 64-bit accumulater with the final result shifted to the required fixed-point format. Therefore overflow behavior of the 32-bit multiply operation and truncation behavior from final shifing of the accumulated multiplication results must be considered for both FIR operations as well as for coefficient step size calculation and FIR coefficient adjustment.
	
	Computing the coefficient adjustment involves taking the reciprocal of the instantaneous power computed by E = x[n]^2 + x[n–1]^2 + ... + x[n–N+1]^2. The reciprocal is subject to overflow since the instantaneous power may be less than one.

Type	```int```
	```lib_dsp_adaptive_nlms(int input_sample,```
	``` int reference_sample,```
	``` int error_sample[],```
	``` int filter_coeffs[],```
	``` int state_data[],```
	``` int tap_count,```
	``` int step_size,```
	``` int q_format)```

Continued on next page

Parameters	`input_sample`
	The new sample to be processed.
	`reference_sample`
	Reference sample.
	`error_sample`
	Pointer to resulting error sample (error = reference - output)
	`filter_coeffs`
	Pointer to FIR coefficients arranged as [b0,b1,b2, ...,bN-1].
	`state_data`
	Pointer to FIR filter state data array of length N. Must be initialized at startup to all zeros.
	`tap_count` Filter tap count where N = tap_count = filter order + 1.
	`step_size` Coefficient adjustment step size, controls rate of convergence.
	`q_format` Fixed point format (i.e. number of fractional bits).
Returns	The resulting filter output sample.

10 Scalar Math Functions: Multiply

Function	lib_dsp_math_multiply

Description	Scalar multipliplication.
	
	This function multiplies two scalar values and produces a result according to fixed-point format specified by the q_format parameter.
	
	The two operands are multiplied to produce a 64-bit result which is tested for overflow, clamped at the minimum/maximum value given the fixed-point format if overflow occurs, and finally shifted right by q_format bits.
	
	Algorithm:
	<pre>1) Y = X1 * X2
	2) Y = min(max(Q_FORMAT_MIN, Y), Q_FORMAT_MAX, Y)
	3) Y = Y >> q_format</pre>
	Example:
	<pre>int result;
	result = lib_dsp_math_multiply(Q28(-0.33), sample, 28);</pre>
	While saturation is employed after multiplication an overflow condition when preparing the final result must still be considered when specifying a Q-format whose fixed-point numerical range do not accomodate the final result of multiplication and saturation (if applied).

Type	<pre>int
	lib_dsp_math_multiply(int input1_value,
	                      int input2_value,
	                      int q_format)</pre>

Parameters	input1_value
	        Multiply operand #1.
	
	input2_value
	        Multiply operand #2.
	
	q_format Fixed point format (i.e. number of fractional bits).

Returns	input1_value * input2_value.

11   Scalar Math Functions: Reciprocal

Function	**lib_dsp_math_reciprocal**

Description	Scalar reciprocal.
	This function computes the reciprocal of the input value using an iterative approximation method as follows:
	
	   1) `result = 1.0`
	   2) `result = result + result * (1 - input * result)`
	   3) `Repeat step #2 until desired precision is achieved`
	
	Example:
	
	   `int result;`
	   `result = lib_dsp_math_reciprocal( sample, 28 );`

Type	`int` `lib_dsp_math_reciprocal(int input_value, int q_format)`
Parameters	`input_value` Input value for computation. `q_format` Fixed point format (i.e. number of fractional bits).
Returns	The reciprocal of the input value.

12 Scalar Math Functions: Inverse Square Root

Function	lib_dsp_math_invsqrroot

Description	Scalar inverse square root.
	This function computes the reciprocal of the square root of the input value using an iterative approximation method as follows:
	
	`1) result = 1.0`
	`2) result = result + result * (1 - input * result^2) / 2`
	`3) Repeat step #2 until desired precision is achieved`
	
	Example:
	
	`int result;`
	`result = lib_dsp_math_invsqrroot( sample, 28 );`

Type	`int` `lib_dsp_math_invsqrroot(int input_value, int q_format)`
Parameters	`input_value` Input value for computation. `q_format` Fixed point format (i.e. number of fractional bits).
Returns	The inverse square root of the input value.

13 Scalar Math Functions: Square Root

Function	**lib_dsp_math_squareroot**

Description	Scalar square root.
	This function computes the square root of the input value using the following steps:
	<pre>int result;
	result = lib_dsp_math_invsqrroot(input)
	result = lib_dsp_math_reciprocal(result)</pre>Example:
	<pre>int result;
	result = lib_dsp_math_squareroot(sample, 28);</pre>

Type	<pre>int
	lib_dsp_math_squareroot(int input_value, int q_format)</pre>

Parameters	`input_value` Input value for computation. `q_format` Fixed point format (i.e. number of fractional bits).
Returns	The square root of the input value.

14 Vector Math Functions: Minimum Value

Function	**lib_dsp_vector_minimum**

Description	Vector Minimum.
	Locate the vector's first occurring minimum value, returning the index of the first occurring minimum value.
	Example:
	<pre>int samples[256];
	int result = lib_dsp_vector_minimum(samples, 256);</pre>

Type	<pre>int
	lib_dsp_vector_minimum(const int input_vector[],
	                       int vector_length)</pre>

Parameters	`input_vector`
	          Pointer to source data array.
	
	`vector_length`
	          Length of the output and input arrays.

Returns	Array index where first minimum value occurs.

15   Vector Math Functions: Maximum Value

Function	**lib_dsp_vector_maximum**

Description	Vector Minimum.
	Locate the vector's first occurring maximum value, returning the index of the first occurring maximum value.
	Example:
	
	```int samples[256];```
	```int result = lib_dsp_vector_maximum(samples, 256);```

Type	```int```
	```lib_dsp_vector_maximum(const int input_vector[],```
	``` int vector_length)```

Parameters	```input_vector```
	            Pointer to source data array.
	
	```vector_length```
	            Length of the output and input arrays.

Returns	Array index where first maximum value occurs.

16 Vector Math Functions: Element Negation

Function	lib_dsp_vector_negate

Description	Vector negation: R[i] = –X[i].
	This function computes the negative value for each input element and sets the corresponding result element to its negative value.
	Each negated element is computed by twos-compliment negation therefore the minimum negative fixed-point value can not be negated to generate its corresponding maximum positive fixed-point value. For example: -Q28(-8.0) will not result in a fixed-point value representing +8.0.
	Example:
	<pre>int samples[256];
	int result[256];
	lib_dsp_vector_negate(samples, result, 256);</pre>

Type	<pre>void
	lib_dsp_vector_negate(const int input_vector_X[],
	                      int result_vector_R[],
	                      int vector_length)</pre>

Parameters	input_vector_X
	            Pointer/reference to source data.
	
	result_vector_R
	            Pointer to the resulting data array.
	
	vector_length
	            Length of the input and output vectors.

17 Vector Math Functions: Element Absolute Value

Function	lib_dsp_vector_abs

Description	Vector absolute value: R[i] = |X[i]|.
	Set each element of the result vector to the absolute value of the corresponding input vector element.
	Example:
	`int samples[256];`
	`int result[256];`
	`lib_dsp_vector_abs( samples, result, 256 );`
	
	If an element is less than zero it is negated to compute its absolute value. Negation is computed via twos-compliment negation therefore the minimum negative fixed-point value can not be negated to generate its corresponding maximum positive fixed-point value. For example: -Q28(-8.0) will not result in a fixed-point value representing +8.0.

Type	`void`
	`lib_dsp_vector_abs(const int input_vector_X[],`
	`                    int result_vector_R[],`
	`                    int vector_length)`

Parameters	`input_vector_X`
	        Pointer/reference to source data.
	
	`result_vector_R`
	        Pointer to the resulting data array.
	
	`vector_length`
	        Length of the input and output vectors.

18 Vector Math Functions: Scalar Addition

Function	lib_dsp_vector_adds

Description	Vector / scalar addition: R[i] = X[i] + A.
	This function adds a scalar value to each vector element.
	32-bit addition is used to compute the scaler plus vector element result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.
	Example:
	<pre>int input_vector_X[256];
	int input_scalar_A = Q28(0.333);
	int result_vector_R[256];
	lib_dsp_vector_adds(input_vector_X, scalar_value_A, result_vector_R,
	     ↪ 256);</pre>

Type	<pre>void
	lib_dsp_vector_adds(const int input_vector_X[],
	                    int input_scalar_A,
	                    int result_vector_R[],
	                    int vector_length)</pre>

Parameters	input_vector_X
	    Pointer/reference to source data array X
	
	input_scalar_A
	    Scalar value to add to each input element
	
	result_vector_R
	    Pointer to the resulting data array
	
	vector_length
	    Length of the input and output vectors

19 Vector Math Functions: Scalar Multiplication

Function	**lib_dsp_vector_muls**

Description	Vector / scalar multiplication: R[i] = X[i] * A.
	32-bit addition is used to compute the scaler plus vector element result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.
	Example:
	<pre>int input_vector_X[256];
	int input_scalar_A = Q28(0.333);
	int result_vector_R[256];
	lib_dsp_vector_adds(input_vector_X, scalar_value_A, result_vector_R,
	  ↪ 256);</pre>

Type	<pre>void
	lib_dsp_vector_muls(const int input_vector_X[],
	                    int input_scalar_A,
	                    int result_vector_R[],
	                    int vector_length,
	                    int q_format)</pre>

Parameters	input_vector_X
	        Pointer/reference to source data array X.
	
	input_scalar_A
	        Scalar value to multiply each element by.
	
	result_vector_R
	        Pointer to the resulting data array.
	
	vector_length
	        Length of the input and output vectors.
	
	q_format Fixed point format, the number of bits making up fractional part.

20 Vector Math Functions: Vector Addition

Function	lib_dsp_vector_addv

Description	Vector / vector addition: R[i] = X[i] + Y[i].
	32-bit addition is used to compute the scaler plus vector element result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.
	Example:
	
	```int input_vector_X[256];```
	```int input_vector_Y[256];```
	```int result_vector_R[256];```
	```lib_dsp_vector_addv(input_vector_X, input_vector_Y, result_vector_R,```
	```↪ 256);```

Type	```void```
	```lib_dsp_vector_addv(const int input_vector_X[],```
	``` const int input_vector_Y[],```
	``` int result_vector_R[],```
	``` int vector_length)```

Parameters	```input_vector_X```
	          Pointer to source data array X.
	
	```input_vector_Y```
	          Pointer to source data array Y.
	
	```result_vector_R```
	          Pointer to the resulting data array.
	
	```vector_length```
	          Length of the input and output vectors.

21   Vector Math Functions: Vector Subtraction

Function	lib_dsp_vector_subv

Description	Vector / vector subtraction: R[i] = X[i] – Y[i].
	32-bit subtraction is used to compute the scaler plus vector element result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.
	Example:
	<pre>int input_vector_X[256];
	int input_vector_Y[256];
	int result_vector_R[256];
	lib_dsp_vector_subv(input_vector_X, input_vector_Y, result_vector_R,
	↪ 256);</pre>

Type	<pre>void
	lib_dsp_vector_subv(const int input_vector_X[],
	                     const int input_vector_Y[],
	                     int result_vector_R[],
	                     int vector_length)</pre>

Parameters	input_vector_X
	        Pointer to source data array X.
	
	input_vector_Y
	        Pointer to source data array Y.
	
	result_vector_R
	        Pointer to the resulting data array.
	
	vector_length
	        Length of the input and output vectors.

22 Vector Math Functions: Vector Multiplication

Function	**lib_dsp_vector_mulv**

Description	Vector / vector multiplication: R[i] = X[i] * Y[i].
	Elements in each of the input vectors are multiplied together using a 32-bit multiply 64-bit accumulate function therefore fixed-point multiplication and q-format adjustment overflow behavior must be considered (see behavior for the function `lib_dsp_math_multiply`).
	Example:
	<pre>int input_vector_X[256];
	int input_vector_Y[256];
	int result_vector_R[256];
	lib_dsp_vector_mulv(input_vector_X, input_vector_Y, result_vector_R,
	  ↪ 256, 28);</pre>

Type	<pre>void
	lib_dsp_vector_mulv(const int input_vector_X[],
	                    const int input_vector_Y[],
	                    int result_vector_R[],
	                    int vector_length,
	                    int q_format)</pre>

Parameters	`input_vector_X`
	       Pointer to source data array X.
	
	`input_vector_Y`
	       Pointer to source data array Y.
	
	`result_vector_R`
	       Pointer to the resulting data array.
	
	`vector_length`
	       Length of the input and output vectors.
	
	`q_format` Fixed point format (i.e. number of fractional bits).

23   Vector Math Functions: Vector multiplication and scalar addition

Function	**lib_dsp_vector_mulv_adds**

Description	Vector multiplication and scalar addition: R[i] = X[i] * Y[i] + A.
	Elements in each of the input vectors are multiplied together using a 32-bit multiply 64-bit accumulate function therefore fixed-point multiplication and q-format adjustment overflow behavior must be considered (see behavior for the function lib_dsp_math_multiply). 32-bit addition is used to compute the vector element plus scalar value result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.
	Example:
	<pre>int input_vector_X[256];
	int input_vector_Y[256];
	int input_scalar_A = Q28(0.333);
	int result_vector_R[256];
	lib_dsp_vector_mulv_adds(input_vector_X, input_vector_Y, scalar_value_A,
	  ↪ result_vector_R, 256, 28);</pre>

Type	<pre>void
	lib_dsp_vector_mulv_adds(const int input_vector_X[],
	    const int input_vector_Y[],
	    int input_scalar_A,
	    int result_vector_R[],
	    int vector_length,
	    int q_format)</pre>

Parameters	input_vector_X
	        Pointer to source data array X.
	
	input_vector_Y
	        Pointer to source data array Y.
	
	input_scalar_A
	        Scalar value to add to each X*Y result.
	
	result_vector_R
	        Pointer to the resulting data array.
	
	vector_length
	        Length of the input and output vectors.
	
	q_format Fixed point format (i.e. number of fractional bits).

24 Vector Math Functions: Scalar multiplication and vector addition

Function	lib_dsp_vector_muls_addv

Description	Scalar multiplication and vector addition: R[i] = X[i] * A + Y[i].
	Each element in the input vectors is multiplied by a scalar using a 32bit multiply 64-bit accumulate function therefore fixed-point multiplication and q-format adjustment overflow behavior must be considered (see behavior for the function lib_dsp_math_multiply). 32-bit addition is used to compute the vector element minus vector element result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.
	Example:
	<pre>int input_vector_X[256];
	int input_scalar_A = Q28(0.333);
	int input_vector_Y[256];
	int result_vector_R[256];
	lib_dsp_vector_muls_addv(input_vector_X, input_scalar_A, input_vector_Y,
	    ↪ result_vector_R, 256, 28);</pre>

Type	<pre>void
	lib_dsp_vector_muls_addv(const int input_vector_X[],
	    int input_scalar_A,
	    const int input_vector_Y[],
	    int result_vector_R[],
	    int vector_length,
	    int q_format)</pre>

Parameters	input_vector_X
	    Pointer to source data array X.
	
	input_scalar_A
	    Scalar value to multiply each element by.
	
	input_vector_Y
	    Pointer to source data array Y
	
	result_vector_R
	    Pointer to the resulting data array.
	
	vector_length
	    Length of the input and output vectors
	
	q_format Fixed point format (i.e. number of fractional bits).

25 Vector Math Functions: Scalar multiplication and vector subtraction

Function	**lib_dsp_vector_muls_subv**

Description	Scalar multiplication and vector subtraction: R[i] = X[i] * A – Y[i].
	Each element in the input vectors is multiplied by a scalar using a 32bit multiply 64-bit accumulate function therefore fixed-point multiplication and q-format adjustment overflow behavior must be considered (see behavior for the function lib_dsp_math_multiply). 32-bit subtraction is used to compute the vector element minus vector element result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.
	Example:
	<pre>int input_vector_X[256];
	int input_scalar_A = Q28(0.333);
	int input_vector_Y[256];
	int result_vector_R[256];
	lib_dsp_vector_muls_subv(input_vector_X, input_scalar_A, input_vector_Y,
	  ↪ result_vector_R, 256, 28);</pre>

Type	<pre>void
	lib_dsp_vector_muls_subv(const int input_vector_X[],
	    int input_scalar_A,
	    const int input_vector_Y[],
	    int result_vector_R[],
	    int vector_length,
	    int q_format)</pre>

Parameters	input_scalar_A
	        Scalar value to multiply each element by.
	
	input_vector_X
	        Pointer to source data array X.
	
	input_vector_Y
	        Pointer to source data array Y.
	
	result_vector_R
	        Pointer to the resulting data array.
	
	vector_length
	        Length of the input and output vectors.
	
	q_format Fixed point format (i.e. number of fractional bits).

26 Vector Math Functions: Vector multiplication and vector addition

Function	**lib_dsp_vector_mulv_addv**

Description	Vector multiplication and vector addition: R[i] = X[i] * Y[i] + Z[i].
	The elements in the input vectors are multiplied before being summed therefore fixed-point multiplication behavior must be considered (see behavior for the function `lib_dsp_math_multiply`). Due to successive 32-bit additions being accumulated using 64-bit arithmetic overflow during the summation process is unlikely. The final value, being effectively the result of a left-shift by `q_format` bits will potentially overflow the final fixed-point value depending on the resulting summed value and the chosen Q-format.
	Example:
	<pre>int input_vector_X[256];
	int input_vector_Y[256];
	int input_vector_Z[256];
	int result_vector_R[256];
	lib_dsp_vector_mulv_subv(input_vector_X, input_vector_Y, input_vector_Z,
	    ↪ result_vector_R, 256);</pre>

Type	<pre>void
	lib_dsp_vector_mulv_addv(const int input_vector_X[],
	    const int input_vector_Y[],
	    const int input_vector_Z[],
	    int result_vector_R[],
	    int vector_length,
	    int q_format)</pre>

Parameters	`input_vector_X`
	Pointer to source data array X.
	
	`input_vector_Y`
	Pointer to source data array Y.
	
	`input_vector_Z`
	Pointer to source data array Z.
	
	`result_vector_R`
	Pointer to the resulting data array.
	
	`vector_length`
	Length of the input and output vectors.
	
	`q_format` Fixed point format (i.e. number of fractional bits).

27 Vector Math Functions: Vector multiplication and vector subtraction

Function	lib_dsp_vector_mulv_subv

Description	Vector multiplication and vector addition: R[i] = X[i] * Y[i] – Z[i].
	The elements in the input vectors are multiplied before being subtracted therefore fixed-point multiplication behavior must be considered (see behavior for the function lib_dsp_math_multiply). Due to successive 32-bit subtractions being accumulated using 64-bit arithmetic overflow during the summation process is unlikely. The final value, being effectively the result of a left-shift by q_format bits will potentially overflow the final fixed-point value depending on the resulting summed value and the chosen Q-format.
	Example:
	
	`int input_vector_X[256];`
	`int input_vector_Y[256];`
	`int input_vector_Z[256];`
	`int result_vector_R[256];`
	`lib_dsp_vector_mulv_subv( input_vector_X, input_vector_Y, input_vector_Z,`
	`  ↪ result_vector_R, 256 );`

Type	`void`
	`lib_dsp_vector_mulv_subv(const int input_vector_X[],`
	`    const int input_vector_Y[],`
	`    const int input_vector_Z[],`
	`    int result_vector_R[],`
	`    int vector_length,`
	`    int q_format)`

Parameters	`input_vector_X`
	         Pointer to source data array X.
	
	`input_vector_Y`
	         Pointer to source data array Y.
	
	`input_vector_Z`
	         Pointer to source data array Z.
	
	`result_vector_R`
	         Pointer to the resulting data array.
	
	`vector_length`
	         Length of the input and output vectors.
	
	`q_format` Fixed point format (i.e. number of fractional bits).

28 Matrix Math Functions: Element Negation

Function	lib_dsp_matrix_negate

Description	Matrix negation: R[i][j] = –X[i][j].
	Each negated element is computed by twos-compliment negation therefore the minimum negative fixed-point value can not be negated to generate its corresponding maximum positive fixed-point value. For example: -Q28(-8.0) will not result in a fixed-point value representing +8.0.
	Example:
	<pre>int samples[8][32];
	int result[8][32];
	lib_dsp_matrix_negate(samples, result, 8, 32);</pre>

Type	<pre>void
	lib_dsp_matrix_negate(const int input_matrix_X[],
	                      int result_matrix_R[],
	                      int row_count,
	                      int column_count)</pre>

Parameters	input_matrix_X
	            Pointer/reference to source data.
	
	result_matrix_R
	            Pointer to the resulting 2-dimensional data array.
	
	row_count Number of rows in input matrix.
	
	column_count
	            Number of columns in input matrix.

29 Matrix Math Functions: Scalar Addition

Function	lib_dsp_matrix_adds
Description	Matrix / scalar addition: R[i][j] = X[i][j] + A. 32-bit addition is used to compute the scaler plus matrix element result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated. Example: ```\nint input_matrix_X[8][32];\nint input_scalar_A = Q28(0.333);\nint result_vector_R[8][32];\nlib_dsp_matrix_adds(input_matrix_X, scalar_matrix_A, result_matrix_R,\n ↪ 8, 32);\n```
Type	```\nvoid\nlib_dsp_matrix_adds(const int input_matrix_X[],\n int input_scalar_A,\n int result_matrix_R[],\n int row_count,\n int column_count)\n```

Parameters	input_matrix_X
	        Pointer/reference to source data.
	
	input_scalar_A
	        Scalar value to add to each input element.
	
	result_matrix_R
	        Pointer to the resulting 2-dimensional data array.
	
	row_count Number of rows in input and output matrices.
	
	column_count
	        Number of columns in input and output matrices.

30 Matrix Math Functions: Scalar Multiplication

Function	lib_dsp_matrix_muls

Description	Matrix / scalar multiplication: R[i][j] = X[i][j] * A.
	Each element of the input matrix is multiplied by a scalar value using a 32bit multiply 64-bit accumulate function therefore fixed-point multiplication and q-format adjustment overflow behavior must be considered (see behavior for the function `lib_dsp_math_multiply`).
	Example:
	<pre>int input_matrix_X[8][32];
	int input_scalar_A = Q28(0.333);
	int result_vector_R[8][32];
	lib_dsp_matrix_muls(input_matrix_X, scalar_value_A, result_matrix_R, 256, 8, 32, 28
	↪);</pre>

Type	<pre>void
	lib_dsp_matrix_muls(const int input_matrix_X[],
	                    int input_scalar_A,
	                    int result_matrix_R[],
	                    int row_count,
	                    int column_count,
	                    int q_format)</pre>

Parameters	`input_matrix_X`
	    Pointer/reference to source data X.
	
	`input_scalar_A`
	    Scalar value to multiply each element by.
	
	`result_matrix_R`
	    Pointer to the resulting 2-dimensional data array.
	
	`row_count` Number of rows in input and output matrices.
	
	`column_count`
	    Number of columns in input and output matrices.
	
	`q_format` Fixed point format (i.e. number of fractional bits).

31 Matrix Math Functions: Matrix Addition

Function	lib_dsp_matrix_addm
Description	Matrix / matrix addition: R[i][j] = X[i][j] + Y[i][j]. 32-bit addition is used to compute the result for each element. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated. Example: ```\nint input_matrix_X [256];\nint input_matrix_Y [256];\nint result_matrix_R[256];\nlib_dsp_matrix_addv(input_matrix_X, input_matrix_Y, result_matrix_R,\n ↪ 8, 32);\n```
Type	```\nvoid\nlib_dsp_matrix_addm(const int input_matrix_X[],\n const int input_matrix_Y[],\n int result_matrix_R[],\n int row_count,\n int column_count)\n```

Parameters	input_matrix_X
	        Pointer to source data array X.
	
	input_matrix_Y
	        Pointer to source data array Y.
	
	result_matrix_R
	        Pointer to the resulting 2-dimensional data array.
	
	row_count Number of rows in input and output matrices.
	
	column_count
	        Number of columns in input and output matrices.

32 Matrix Math Functions: Matrix Subtraction

Function	**lib_dsp_matrix_subm**

Description	Matrix / matrix subtraction: R[i][j] = X[i][j] – Y[i][j].
	32-bit subtraction is used to compute the result for each element. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.
	Example:
	<pre>int input_matrix_X [256];
	int input_matrix_Y [256];
	int result_matrix_R[256];
	lib_dsp_matrix_addv(input_matrix_X, input_matrix_Y, result_matrix_R,
	↪ 8, 32);</pre>

Type	<pre>void
	lib_dsp_matrix_subm(const int input_matrix_X[],
	                    const int input_matrix_Y[],
	                    int result_matrix_R[],
	                    int row_count,
	                    int column_count)</pre>

Parameters	input_matrix_X
	          Pointer to source data array X.
	
	input_matrix_Y
	          Pointer to source data array Y.
	
	result_matrix_R
	          Pointer to the resulting 2-dimensional data array.
	
	row_count Number of rows in input and output matrices.
	
	column_count
	          Number of columns in input and output matrices.

33 Matrix Math Functions: Matrix Multiplication

Function	lib_dsp_matrix_mulm
Description	Matrix / matrix multiplication: R[i][j] = X[i][j] * Y[i][j]. Elements in each of the input matrices are multiplied together using a 32bit multiply 64-bit accumulate function therefore fixed-point multiplication and q-format adjustment overflow behavior must be considered (see behavior for the function lib_dsp_math_multiply). Example: ```int input_matrix_X[8][32];\nint input_matrix_Y[8][32];\nint result_vector_R[8][32];\nlib_dsp_matrix_mulm(input_matrix_X, input_matrix_Y, result_matrix_R, 256, 8, 32, 28\n↪);```
Type	```void\nlib_dsp_matrix_mulm(const int input_matrix_X[],\n const int input_matrix_Y[],\n int result_matrix_R[],\n int row_count,\n int column_count,\n int q_format)```

Parameters	input_matrix_X
	    Pointer to source data array X.
	
	input_matrix_Y
	    Pointer to source data array Y.
	
	result_matrix_R
	    Pointer to the resulting 2-dimensional data array.
	
	row_count Number of rows in input and output matrices.
	
	column_count
	    Number of columns in input and output matrices.
	
	q_format Fixed point format (i.e. number of fractional bits).

34 Statistics Functions: Vector Mean

Function	lib_dsp_vector_mean

Description	Vector mean: R = (X[0] + X[N–1]) / N.
	This function computes the mean of the values contained within the input vector. Due to successive 32-bit additions being accumulated using 64-bit arithmetic overflow during the summation process is unlikely. The final value, being effectively the result of a left-shift by q_format bits will potentially overflow the final fixed-point value depending on the resulting summed value and the chosen Q-format.
	Example:
	<pre>int result;
	result = lib_dsp_vector_mean(input_vector, 256, 28);</pre>

Type	<pre>int
	lib_dsp_vector_mean(const int input_vector_X[],
	                    int vector_length,
	                    int q_format)</pre>

Parameters	`input_vector_X`
	    Pointer to source data array X.
	
	`vector_length`
	    Length of the input vector.
	
	`q_format` Fixed point format (i.e. number of fractional bits).

35 Statistics Functions: Vector Power (Sum-of-Squares)

Function	**lib_dsp_vector_power**

Description	Vector power (sum of squares): $R = X[0]^2 + X[N-1]^2$.
	 This function computes the power (also know as the sum-of-squares) of the values contained within the input vector.
	 Since each element in the vector is squared the behavior for fixed-point multiplication should be considered (see behavior for the function `lib_dsp_math_multiply`). Due to successive 32-bit additions being accumulated using 64-bit arithmetic overflow during the summation process is unlikely. The final value, being effectively the result of a left-shift by q_format bits will potentially overflow the final fixed-point value depending on the resulting summed value and the chosen Q-format.
	 Example:
	
	`int result;`
	`result = lib_dsp_vector_power( input_vector, 256, 28 );`

Type	`int`
	`lib_dsp_vector_power(const int input_vector_X[],`
	`                      int vector_length,`
	`                      int q_format)`

Parameters	`input_vector_X`
	　　　　　Pointer to source data array X.
	
	`vector_length`
	　　　　　Length of the input vector.
	
	`q_format`　　　Fixed point format (i.e. number of fractional bits).

36 Statistics Functions: Root Mean Square (RMS)

Function	lib_dsp_vector_rms

Description	Vector root mean square: R = ((X[0]^2 + X[N–1]^2) / N) ^ 0.5).
	This function computes the root-mean-square (RMS) of the values contained within the input vector.
	<pre>result = 0
	for i = 0 to N-1: result += input_vector_X[i]
	return lib_dsp_math_squareroot(result / vector_length)</pre>
	Since each element in the vector is squared the behavior for fixed-point multiplication should be considered (see behavior for the function `lib_dsp_math_multiply`). Due to successive 32-bit additions being accumulated using 64-bit arithmetic overflow during the summation process is unlikely. The squareroot of the 'sum-of-squares divided by N uses the function `lib_dsp_math_squareroot`; see behavior for that function. The final value, being effectively the result of a left-shift by `q_format` bits will potentially overflow the final fixed-point value depending on the resulting summed value and the chosen Q-format.
	Example:
	<pre>int result;
	result = lib_dsp_vector_rms(input_vector, 256, 28);</pre>

Type	<pre>int
	lib_dsp_vector_rms(const int input_vector_X[],
	                   int vector_length,
	                   int q_format)</pre>

Parameters	`input_vector_X`
	        Pointer to source data array X.
	
	`vector_length`
	        Length (N) of the input vector.
	
	`q_format` Fixed point format (i.e. number of fractional bits).

37 Statistics Functions: Dot Product

Function	lib_dsp_vector_dotprod

Description	Vector dot product: R = X[0] * Y[0] + X[N-1] * Y[N-1].
	This function computes the dot-product of two equal length vectors.
	The elements in the input vectors are multiplied before being summed therefore fixed-point multiplication behavior must be considered (see behavior for the function `lib_dsp_math_multiply`). Due to successive 32-bit additions being accumulated using 64-bit arithmetic overflow during the summation process is unlikely. The final value, being effectively the result of a left-shift by q_format bits will potentially overflow the final fixed-point value depending on the resulting summed value and the chosen Q-format.
	Example:
	
	`int result;`
	`result = lib_dsp_vector_dotprod( input_vector, 256, 28 );`

Type	`int`
	`lib_dsp_vector_dotprod(const int input_vector_X[],`
	`                        const int input_vector_Y[],`
	`                        int vector_length,`
	`                        int q_format)`

Parameters	`input_vector_X`
	        Pointer to source data array X.
	
	`input_vector_Y`
	        Pointer to source data array Y.
	
	`vector_length`
	        Length of the input vectors.
	
	`q_format` Fixed point format (i.e. number of fractional bits).

38 Transform Functions: FFT Bit Reversal

Function	lib_dsp_fft_bit_reverse
Description	This function preforms index bit reversing on the the arrays around prior to computing an FFT. A calling sequence for a forward FFT involves lib_dsp_fft_bit_reverse() followed by lib_dsp_fft_forward_complex(), and for an inverse FFT it involves lib_dsp_fft_bit_reverse() followed by lib_dsp_fft_inverse_complex(). In some cases bit reversal can be avoided, for example when computing a convolution.
Type	`void` `lib_dsp_fft_bit_reverse(lib_dsp_fft_complex_t pts[], int N)`
Parameters	pts Array of complex input points N number of points. Must be a power of 2, both re and im should be N long

39 Transform Functions: Forward Complex FFT

Function	lib_dsp_fft_forward_complex
Description	This function computes a forward FFT. The complex input signal is supplied in an array of real and imaginary fixed-point values. The same array is also used to store the output. The number of points must be a power of 2, and the array of sine values should contain a quarter sine-wave. Use one of the lib_dsp_sine_N tables. The function does not perform bit reversal of the data. If required then lib_dsp_fft_bit_reverse() should be called beforehand.
Type	```void lib_dsp_fft_forward_complex(lib_dsp_fft_complex_t pts[], int N, const int sine[])```
Parameters	pts Array of complex input points. Must be 64-bit aligned to enable efficient double word loads and stores N number of points. Must be a power of 2. sine array of N/4+1 sine values, each represented as a sign bit, and a 31 bit fraction. 1 should be represented as 0x7fffffff. Arrays are provided in lib_dsp_tables.c; for example, for a 1024 point FFT use lib_dsp_sine_1024.

40 Transform Functions: Forward Real FFT's

Function	lib_dsp_fft_forward_tworeals
Description	This function computes the FFT of two real sequences in one go. It uses a nifty trick () that enables one to use a single complex FFT to compute two real FFTs simultaneously. The real inputs should be in the first two real arrays, the output is in the real and imaginary arrays (the output of a real FFT is still a complex number).
Type	```void lib_dsp_fft_forward_tworeals(lib_dsp_fft_complex_t two_re[], lib_dsp_fft_complex_t two_im[], int N, const int sine[])```
Parameters	two_re array containing two real signals on which to compute FFT, on output this array stores the real part of the complex FFT on the two real signals. two_im imaginary parts of complex FFT of two real signals in first array N number of points sine array of N/4+1 sine values, each represented as a sign bit, and a 31 bit fraction. 1 should be represented as 0x7fffffff. Arrays are provided in lib_dsp_tables.c; for example, for a 1024 point FFT use lib_dsp_sine_1024.

41 Transform Functions: Inverse Complex FFT

Function	lib_dsp_fft_inverse_complex
Description	This function computes an inverse FFT. The complex input array is supplied as two arrays of integers, with numbers represented as fixed-point values. The number of points must be a power of 2, and the array of sine values should contain a quarter sine-wave. Use one of the lib_dsp_sine_N tables. The function does not perform a bit-twiddle - if required then lib_dsp_fft_bit_reverse() should be called beforehand.
Type	```void lib_dsp_fft_inverse_complex(lib_dsp_fft_complex_t pts[], int N, const int sine[])```
Parameters	pts Array of complex input points N number of points. Must be a power of 2, both re and im should be N long sine array of N/4+1 sine values, each represented as a sign bit, and a 31 bit fraction. 1 should be represented as 0x7fffffff. Arrays are provided in lib_dsp_tables.c; for example, for a 1024 point FFT use lib_dsp_sine_1024.

42   Transform Functions: Inverse Real FFT's

Function	lib_dsp_fft_inverse_tworeals
Description	This function computes the inverse FFT on two sets of complex data that are known to result in real numbers only in one go. It uses a nifty trick () that enables one to use a single complex inverse FFT to compute two real inverse FFTs simultaneously. The outputs are in the two real arrays, the imaginary arrays are unchanged.

Type	```void
	lib_dsp_fft_inverse_tworeals(lib_dsp_fft_complex_t two_re[],
	    lib_dsp_fft_complex_t two_im[],
	    int N,
	    const int sine[])```

Parameters	two_re array containing two real frequency domain signals on which to compute inverse FFT. On output this array stores two real time domain signals two_im array containing two imaginary frequency domain signals on which to compute inverse FFT N number of points sine array of N/4+1 sine values, each represented as a sign bit, and a 31 bit fraction. 1 should be represented as 0x7fffffff. Arrays are provided in lib_dsp_tables.c; for example, for a 1024 point FFT use lib_dsp_sine_1024.

43 Filter Design Functions: Notch Filter

Function	lib_dsp_design_biquad_notch

Description	This function generates BiQuad filter coefficients for a notch filter.
	The filter coefficients are stored in forward order (e.g. b0,b1,b2,a1,a2). The frequency specification is normalized to the Nyquist frequency therefore the frequency value must be in the range of 0.0 <= F < 0.5 for valid filter coefficients.
	Example showing a filter coefficients generation using Q28 fixed-point formatting.
	<pre>int coeffs[5];
	lib_dsp_design_biquad_notch(0.25, 0.707, coeffs, 28);</pre>

Type	<pre>void
	lib_dsp_design_biquad_notch(double filter_frequency,
	                            double filter_Q,
	                            int biquad_coeffs[5],
	                            int q_format)</pre>

Parameters	filter_frequency
	        Filter center frequency normalized to the sampling frequency. 0 < frequency < 0.5, where 0.5 represents Fs/2.
	
	filter_Q The filter Q-factor.
	
	biquad_coeffs
	        The array used to contain the resulting filter coefficients. Filter coefficients are ordered as [b0,b1,b2,a1,a2].
	
	q_format Fixed point format of coefficients (i.e. number of fractional bits).

44   Filter Design Functions: Low-pass Filter

Function	**lib_dsp_design_biquad_lowpass**

Description	This function generates BiQuad filter coefficients for a low-pass filter.
	The filter coefficients are stored in forward order (e.g. b0,b1,b2,a1,a2). The frequency specification is normalized to the Nyquist frequency therefore the frequency value must be in the range of `0.0 <= F < 0.5` for valid filter coefficients.
	Example showing a filter coefficients generation using Q28 fixed-point formatting.
	
	`int coeffs[5];`
	`lib_dsp_design_biquad_lowpass( 0.25, 0.707, coeffs, 28 );`

Type	`void`
	`lib_dsp_design_biquad_lowpass(double filter_frequency,`
	`                                double filter_Q,`
	`                                int biquad_coeffs[5],`
	`                                int q_format)`

Parameters	`filter_frequency`
	          Filter cutoff (-3db) frequency normalized to the sampling frequency.
	          `0 < frequency < 0.5`, where 0.5 represents Fs/2.
	
	`filter_Q` The filter Q-factor.
	
	`biquad_coeffs`
	          The array used to contain the resulting filter coefficients. Filter coefficients are ordered as `[b0,b1,b2,a1,a2]`.
	
	`q_format` Fixed point format of coefficients (i.e. number of fractional bits).

45   Filter Design Functions: High-pass Filter

Function	**lib_dsp_design_biquad_highpass**

Description	This function generates BiQuad filter coefficients for a high-pass filter.
	The filter coefficients are stored in forward order (e.g. b0,b1,b2,a1,a2). The frequency specification is normalized to the Nyquist frequency therefore the frequency value must be in the range of `0.0 <= F < 0.5` for valid filter coefficients.
	Example showing a filter coefficients generation using Q28 fixed-point formatting.
	
	```int coeffs[5];```
	```lib_dsp_design_biquad_highpass(0.25, 0.707, coeffs, 28);```

Type	```void```
	```lib_dsp_design_biquad_highpass(double filter_frequency,```
	``` double filter_Q,```
	``` int biquad_coeffs[5],```
	``` int q_format)```

Parameters	```filter_frequency```
	Filter cutoff (-3db) frequency normalized to the sampling frequency. 0 < frequency < 0.5, where 0.5 represents Fs/2.
	
	```filter_Q``` The filter Q-factor.
	
	```biquad_coeffs```
	The array used to contain the resulting filter coefficients. Filter coefficients are ordered as [b0,b1,b2,a1,a2].
	
	```q_format``` Fixed point format of coefficients (i.e. number of fractional bits).

46   Filter Design Functions: All-pass Filter

Function	**lib_dsp_design_biquad_allpass**

Description	This function generates BiQuad filter coefficients for an all-pass filter.
	The filter coefficients are stored in forward order (e.g. b0,b1,b2,a1,a2). The frequency specification is normalized to the Nyquist frequency therefore the frequency value must be in the range of `0.0 <= F < 0.5` for valid filter coefficients.
	Example showing a filter coefficients generation using Q28 fixed-point formatting.
	<pre>int coeffs[5];
	lib_dsp_design_biquad_allpass(0.25, 0.707, coeffs, 28);</pre>

Type	<pre>void
	lib_dsp_design_biquad_allpass(double filter_frequency,
	                             double filter_Q,
	                             int biquad_coeffs[5],
	                             int q_format)</pre>

Parameters	`filter_frequency`
	    Filter center frequency normalized to the sampling frequency. 0 < frequency < 0.5, where 0.5 represents Fs/2.
	
	`filter_Q` The filter Q-factor.
	
	`biquad_coeffs`
	    The array used to contain the resulting filter coefficients. Filter coefficients are ordered as [b0,b1,b2,a1,a2].
	
	`q_format` Fixed point format of coefficients (i.e. number of fractional bits).

47 Filter Design Functions: Band-pass Filter

Function	**lib_dsp_design_biquad_bandpass**
Description	This function generates BiQuad filter coefficients for a band-pass filter. The filter coefficients are stored in forward order (e.g. b0,b1,b2,a1,a2). The frequency specification is normalized to the Nyquist frequency therefore the frequency value must be in the range of 0.0 <= F < 0.5 for valid filter coefficients. Example showing a filter coefficients generation using Q28 fixed-point formatting. ```\nint coeffs[5];\nlib_dsp_design_biquad_bandpass(0.20, 0.30, coeffs, 28);\n```
Type	```\nvoid\nlib_dsp_design_biquad_bandpass(double filter_frequency1,\n double filter_frequency2,\n int biquad_coeffs[5],\n int q_format)\n```

Parameters	filter_frequency1
	        Filter cutoff #1 (-3db) frequency normalized to the sampling frequency. 0 < frequency1 < 0.5, where 0.5 represents Fs/2.
	
	filter_frequency2
	        Filter cutoff #2 (-3db) frequency normalized to the sampling frequency. 0 < frequency2 < 0.5, where 0.5 represents Fs/2. Note that frequency1 must be less than to frequency2.
	
	biquad_coeffs
	        The array used to contain the resulting filter coefficients. Filter coefficients are ordered as [b0,b1,b2,a1,a2].
	
	q_format Fixed point format of coefficients (i.e. number of fractional bits).

48 Filter Design Functions: Peaking Filter

Function	**lib_dsp_design_biquad_peaking**

Description	This function generates BiQuad filter coefficients for a peaking filter.
	The filter coefficients are stored in forward order (e.g. b0,b1,b2,a1,a2). The frequency specification is normalized to the Nyquist frequency therefore the frequency value must be in the range of `0.0 <= F < 0.5` for valid filter coefficients.
	Example showing a filter coefficients generation using Q28 fixed-point formatting.
	
	`int coeffs[5];`
	`lib_dsp_design_biquad_notch( 0.25, 0.707, coeffs, 28 );`

Type	`void`
	`lib_dsp_design_biquad_peaking(double filter_frequency,`
	`                                double filter_Q,`
	`                                double peak_qain_db,`
	`                                int biquad_coeffs[5],`
	`                                int q_format)`

Parameters	`filter_frequency`
	    Filter center frequency normalized to the sampling frequency. 0 < frequency < 0.5, where 0.5 represents Fs/2.
	
	`filter_Q` The filter Q-factor.
	
	`peak_qain_db`
	    The filter gain in dB (postive or negative). +gain results in peaking gain (gain at peak center = gain_db). -gain results in attenuation (gain at peak center = -gain_db).
	
	`biquad_coeffs`
	    The array used to contain the resulting filter coefficients. Filter coefficients are ordered as [b0,b1,b2,a1,a2].
	
	`q_format` Fixed point format of coefficients (i.e. number of fractional bits).

49 Filter Design Functions: Base Shelving Filter

Function	**lib_dsp_design_biquad_lowshelf**

Description	This function generates BiQuad filter coefficients for a bass shelving filter. The filter coefficients are stored in forward order (e.g. b0,b1,b2,a1,a2). The frequency specification is normalized to the Nyquist frequency therefore the frequency value must be in the range of `0.0 <= F < 0.5` for valid filter coefficients. Example showing a filter coefficients generation using Q28 fixed-point formatting.
	
	`int coeffs[5];`
	`lib_dsp_design_biquad_lowshelf( 0.25, 0.707, +6.0, coeffs, 28 );`

Type	`void`
	`lib_dsp_design_biquad_lowshelf(double filter_frequency,`
	`                              double filter_Q,`
	`                              double shelf_gain_db,`
	`                              int biquad_coeffs[5],`
	`                              int q_format)`

Parameters	`filter_frequency`
	    Filter frequency (+3db or -3db point) normalized to Fs. 0 < frequency < 0.5, where 0.5 represents Fs/2.
	
	`filter_Q` The filter Q-factor.
	
	`shelf_gain_db`
	    The filter shelf gain in dB (postive or negative). +gain results in bass shelf with gain of 'shelf_gain_db'. -gain results in bass shelf with attenuation of 'shelf_gain_db'.
	
	`biquad_coeffs`
	    The array used to contain the resulting filter coefficients. Filter coefficients are ordered as [b0,b1,b2,a1,a2].
	
	`q_format` Fixed point format of coefficients (i.e. number of fractional bits).

50 Filter Design Functions: Treble Shelving Filter

Function	**lib_dsp_design_biquad_highshelf**

Description	This function generates BiQuad filter coefficients for a treble shelving filter. The filter coefficients are stored in forward order (e.g. b0,b1,b2,a1,a2). The frequency specification is normalized to the Nyquist frequency therefore the frequency value must be in the range of `0.0 <= F < 0.5` for valid filter coefficients. Example showing a filter coefficients generation using Q28 fixed-point formatting.
	
	```int coeffs[5];```
	```lib_dsp_design_biquad_lowshelf(0.25, 0.707, +6.0, coeffs, 28);```

Type	```void```
	```lib_dsp_design_biquad_highshelf(double filter_frequency,```
	``` double filter_Q,```
	``` double shelf_gain_db,```
	``` int biquad_coeffs[5],```
	``` int q_format)```

Parameters	`filter_frequency`
	    Filter frequency (+3db or -3db point) normalized to Fs. 0 < frequency < 0.5, where 0.5 represents Fs/2.
	
	`filter_Q` The filter Q-factor.
	
	`shelf_gain_db`
	    The filter shelf gain in dB (postive or negative). +gain results in bass shelf with gain of 'shelf_gain_db'. -gain results in bass shelf with attenuation of 'shelf_gain_db'.
	
	`biquad_coeffs`
	    The array used to contain the resulting filter coefficients. Filter coefficients are ordered as [b0,b1,b2,a1,a2].
	
	`q_format` Fixed point format of coefficients (i.e. number of fractional bits).

APPENDIX A - Known Issues

There are no known issues.

APPENDIX B - xCORE-200 DSP library change log

B.1   1.0.3

- Update to source code license and copyright

B.2   1.0.2

- FFT and inverse FFT for two complex short int signals

B.3   1.0.1

- FFT and inverse FFT for complex signals or two real signals.

B.4   1.0.0

- Initial version