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 14.2.1 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 3.1.0 of this library. It is known to work on version 14.3.0 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	dsp_qformat	Q8 through Q31 formats, fixed and floating point conversions
Filters	dsp_filters	FIR, biquad, cascaded biquad, and convolution
Adaptive	dsp_adaptive	LMS and NLMS Adaptive filters
Scalar math	dsp_math	Multiply, divide, square root, exponential, natural logarithm trigonometric, hyperbolic
Vector math	dsp_vector	Scalar/vector add/subtract/multiply, dot product
Matrix math	dsp_matrix	Scalar/matrix add/subtract/multiply, inverse and transpose
Statistics	dsp_statistics	Vector mean, sum-of-squares, root-mean-square, variance
Design	dsp_design	Biquad coefficient generation for various filter types
FFT	dsp_fft	Forward and inverse Fast Fourier Transforms.
Sample Rate Conversion	dsp_ds3, dsp_os3	Downsample by factor of 3 (e.g. 48KHz to 16KHz Audio SRC). Oversample by factor of 3 (e.g. 16KHz to 48KHz Audio SRC).

Table 1: DSP library organization

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 (Q8 through Q31), for all functions except for sample rate conversion, which uses a fixed Q31 format.

Further details about Q Format numbers is available here[1].

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 = 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 = dsp_math_multiply( input1_q30, input2_q29, 31 );

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

[1] https://en.wikipedia.org/wiki/Q_(number_format)

3 Filter Functions: Finite Impulse Response (FIR) Filter

Function	**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 `b[i]` is multiplied by a state variable which equals a previous input sample, or `y[n] = x[n]*b0 + x[n-1]*b1 + x[n-2]*b2 + ... + x[n-N+1]*bN-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. `b0,b1,...,bN-1`).
	The following example shows a five-tap (4th order) FIR filter with samples and coefficients represented in Q28 fixed-point format.
	`int32_t filter_coeff[5] = { Q28(0.5),Q28(-0.5),Q28(0.0),Q28(-0.5),Q28(0.5) ↪ };`
	`int32_t filter_state[4] = { 0, 0, 0, 0 };`
	`int32_t result = dsp_filters_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 a 64-bit accumulator. If overflow occurs in the final 64-bit result, it is saturated at the minimum/maximum value given the fixed-point format and finally shifted right by `q_format` bits. The saturation is only done after the last multiplication. To avoid 64-bit overflow in the intermediate results, the fixed point format must be chosen according to num_taps.

Type	`int32_t`
	`dsp_filters_fir(int32_t input_sample,`
	`                const int32_t filter_coeffs[],`
	`                int32_t state_data[],`
	`                const int32_t num_taps,`
	`                const int32_t 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-1. Must be initialized at startup to all zeros. `num_taps` Number of filter taps (N = num_taps = 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	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 `b[i]` is multiplied by a state variable which equals a previous input sample, or `y[n] = x[n]*b0 + x[n-1]*b1 + x[n-2]*b2 + ... + x[n-N+1]*bN-1` `filter_coeffs` points to a coefficient array of size N = num_taps. The filter coefficients are stored in forward order (e.g. `b0,b1,...,bN-1`). Multiplication results are accumulated in a 64-bit accumulator. If overflow occurs in the final 64-bit result, it is saturated at the minimum/maximum value given the fixed-point format and finally shifted right by `q_format` bits. The saturation is only done after the last multiplication. To avoid 64-bit overflow in the intermediate results, the fixed point format must be chosen according to num_taps.

Type	`void`
	`dsp_filters_interpolate(int32_t input_sample,`
	`                        const int32_t filter_coeffs[],`
	`                        int32_t state_data[],`
	`                        const int32_t num_taps,`
	`                        const int32_t interp_factor,`
	`                        int32_t output_samples[],`
	`                        const int32_t q_format)`

Continued on next page

Parameters		
	`input_sample`	The new sample to be processed.
	`filter_coeffs`	Pointer to FIR coefficients array arranged as: `bM,b(1L+M),b(2L+M),b((N-1)L+M),` `b1,b(1L+1),b(2L+1),b((N-1)L+1),` `b0,b(1L+0),b(2L+0),b((N-1)L+0),` where M = N-1
	`state_data`	Pointer to filter state data array of length N-1. Must be initialized at startup to all zeros.
	`num_taps`	Number of filter taps (N = num_taps = 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	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 `b[i]` is multiplied by a state variable which equals a previous input sample, or `y[n] = x[n]*b0 + x[n-1]*b1 + x[n-2]*b2 + ... + x[n-N+1]*bN-1` `filter_coeffs` points to a coefficient array of size N = num_taps. The filter coefficients are stored in forward order (e.g. `b0,b1,...,bN-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 a 64-bit accumulator. If overflow occurs in the final 64-bit result, it is saturated at the minimum/maximum value given the fixed-point format and finally shifted right by `q_format` bits. The saturation is only done after the last multiplication. To avoid 64-bit overflow in the intermediate results, the fixed point format must be chosen according to num_taps.

Type	`int32_t`
	`dsp_filters_decimate(int32_t input_samples[],`
	`const int32_t filter_coeffs[],`
	`int32_t state_data[],`
	`const int32_t num_taps,`
	`const int32_t decim_factor,`
	`const int32_t 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-1. Must be initialized at startup to all zeros. `num_taps` Number of filter taps (N = num_taps = 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	dsp_filters_biquad

Description	This function implements a second order IIR filter (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 executes a difference equation on current and past input values x and past output values y: `y[n] = x[n]*b0 + x[n-1]*b1 + x[n-2]*b2 + y[n-1]*-a1 + y[n-2]*-a2`
	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:
	`int32_t filter_coeff[DSP_NUM_COEFFS_PER_BIQUAD] = { Q28(+0.5), Q28(-0.1), Q28(-0.5), ↪ Q28(-0.1), Q28(0.1) };`
	`int32_t filter_state[DSP_NUM_STATES_PER_BIQUAD] = { 0, 0, 0, 0 };`
	`int32_t result = dsp_filters_biquad( sample, filter_coeff, filter_state, 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 a 64-bit accumulator. If overflow occurs in the final 64-bit result, it is saturated at the minimum/maximum value given the fixed-point format and finally shifted right by q_format bits.

Type	`int32_t`
	`dsp_filters_biquad(int32_t input_sample,`
	`   const int32_t filter_coeffs[DSP_NUM_COEFFS_PER_BIQUAD],`
	`   int32_t state_data[DSP_NUM_STATES_PER_BIQUAD],`
	`   const int32_t q_format)`

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	**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 executes a difference equation on current and past input values x and past output values y: `y[n] = x[n]*b0 + x[n-1]*b1 + x[n-2]*b2 + y[n-1]*-a1 + y[n-2]*-a2`
	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:
	`int32_t filter_coeff[4*DSP_NUM_COEFFS_PER_BIQUAD] = {`
	`    Q28(+0.5), Q28(-0.1), Q28(-0.5), Q28(-0.1), Q28(0.1),`
	`    Q28(+0.5), Q28(-0.1), Q28(-0.5), Q28(-0.1), Q28(0.1),`
	`    Q28(+0.5), Q28(-0.1), Q28(-0.5), Q28(-0.1), Q28(0.1),`
	`    Q28(+0.5), Q28(-0.1), Q28(-0.5), Q28(-0.1), Q28(0.1) };`
	`int32_t filter_state[4*DSP_NUM_STATES_PER_BIQUAD] = { 0,0,0,0, 0,0,0,0, 0,0,0,0, ↪ 0,0,0,0 };`
	`int32_t result = dsp_filters_biquads( 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 a 64-bit accumulator. If overflow occurs in the final 64-bit result, it is saturated at the minimum/maximum value given the fixed-point format and finally shifted right by q_format bits.

Type	`int32_t`
	`dsp_filters_biquads(int32_t input_sample,`
	`                    const int32_t filter_coeffs[],`
	`                    int32_t state_data[],`
	`                    const int32_t num_sections,`
	`                    const int32_t 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	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:
	`int32_t filter_coeff[100] = { ... not shown for brevity };`
	`int32_t filter_state[100] = { 0, 0, 0, 0, ... not shown for brevity };`
	
	`int32_t output_sample = 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 a FIR filter as well as coefficient step size calculations.
	Multiplication results are accumulated in a 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 for both FIR operations as well as for coefficient step size calculation and FIR coefficient adjustment.

Type	`int32_t`
	`dsp_adaptive_lms(int32_t input_sample,`
	`                 int32_t reference_sample,`
	`                 int32_t *error_sample,`
	`                 const int32_t filter_coeffs[],`
	`                 int32_t state_data[],`
	`                 const int32_t num_taps,`
	`                 const int32_t mu,`
	`                 int32_t 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. `num_taps` Filter tap count where N = num_taps = filter order + 1. `mu` 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	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:
	`int32_t filter_coeff[100] = { ... not shown for brevity };`
	`int32_t filter_state[100] = { 0, 0, 0, 0, ... not shown for brevity };`
	
	`int32_t output_sample = 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 a FIR filter as well as coefficient step size calculations.
	Multiplication results are accumulated in a 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 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.

Continued on next page

Type	`int32_t`
	`dsp_adaptive_nlms(int32_t input_sample,`
	`                  int32_t reference_sample,`
	`                  int32_t *error_sample,`
	`                  const int32_t filter_coeffs[],`
	`                  int32_t state_data[],`
	`                  const int32_t num_taps,`
	`                  const int32_t mu,`
	`                  int32_t q_format)`

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. `num_taps` Filter tap count where N = num_taps = filter order + 1. `mu` 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	**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, and shifted right by `q_format` bits.
	Algorithm:
	`1) Y = X1 * X2`
	`3) Y = Y >> q_format`
	Example:
	`int32_t  result;`
	`result = dsp_math_multiply( Q28(-0.33), sample, 28 );`

Type	`int32_t`
	`dsp_math_multiply(int32_t input1_value,`
	`                  int32_t input2_value,`
	`                  const int32_t q_format)`

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: Multiply Saturated

Function	**dsp_math_multiply_sat**

Description	Scalar saturated 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, saturated at the minimum/maximum value given the fixed-point format if overflow occurs, and finally shifted right by q_format bits.
	Algorithm:
	`1) Y = X1 * X2`
	`2) Y = min( max( Q_FORMAT_MIN, Y ), Q_FORMAT_MAX, Y )`
	`3) Y = Y >> q_format`
	Example:
	`int32_t  result;`
	`result = dsp_math_multiply_sat( Q28(-0.33), sample, 28 );`

Type	`int32_t`
	`dsp_math_multiply_sat(int32_t input1_value,`
	`                      int32_t input2_value,`
	`                      const int32_t q_format)`

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.

12 Scalar Math Functions: Signed Division

Function	**dsp_math_divide**

Description	Signed Division.
	This function divides two signed integer values and produces a result according to fixed-point format specified by the `q_format` parameter. It was optimised for performance using a dedicated instruction for unsinged long division.
	Example:
	`uint32_t  quotient;`
	`quotient = dsp_math_divide(divident, divisor, 24);`

Type	`int32_t dsp_math_divide(int32_t dividend,`
	`int32_t divisor,`
	`uint32_t q_format)`

Parameters	`dividend` Value to be divided `divisor` Dividing value `q_format` Fixed point32_t format (i.e. number of fractional bits).
Returns	Quotient of dividend/divisor

13 Scalar Math Functions: Unsigned Division

Function	**dsp_math_divide_unsigned**

Description	Unsigned Division.
	This function divides two unsigned integer values and produces a result according to fixed-point format specified by the q_format parameter. It was optimised for performance using a dedicated instruction for unsinged long division.
	Example:
	`uint32_t  quotient;`
	`quotient = dsp_math_divide_unsigned(divident, divisor, 24);`

Type	`uint32_t`
	`dsp_math_divide_unsigned(uint32_t dividend,`
	`                         uint32_t divisor,`
	`                         uint32_t q_format)`

Parameters	`dividend` Value to be divided `divisor` Dividing value `q_format` Fixed point32_t format (i.e. number of fractional bits).
Returns	Quotient of dividend/divisor

14 Scalar Math Functions: Square Root

Function	**dsp_math_sqrt**
Description	Scalar square root. This function computes the square root of an unsigned input value using the Babylonian method of successive averaging. Error is <= 1 LSB and worst case performance is 96 cycles.
Type	`uq8_24 dsp_math_sqrt(uq8_24 x)`
Parameters	x Unsigned 32-bit value in Q8.24 format
Returns	Unsigned 32-bit value in Q8.24 format

15 Scalar Math Functions: Sine

Function	**dsp_math_sin**
Description	This function returns the sine of a q8_24 fixed point number in radians. The input number has to be in the range -MIN_Q8_24 + PI and MIN_Q8_24 - PI.
Type	`q8_24 dsp_math_sin(q8_24 rad)`
Parameters	`rad` input value in radians
Returns	sine(rad)

16 Scalar Math Functions: Cosine

Function	**dsp_math_cos**
Description	This function returns the cosine of a q8_24 fixed point number in radians. The input number has to be in the range -MIN_Q8_24 + PI and MIN_Q8_24 - PI.
Type	`q8_24 dsp_math_cos(q8_24 rad)`
Parameters	`rad` input value in radians
Returns	cosine(rad)

17 Scalar Math Functions: Arctangent

Function	**dsp_math_atan**

Description	This function returns the arctangent of x.
	It uses an algorithm based on Cody and Waite pp. 194-216. The algorihm was optimised for accuracy (using improved precision and rounding) and performance (using a dedicated instruction for unsinged long division) Error compared to atan from math.h is <= 1 LSB. Performance is 84 cycles worst case. MIN_INT is an invalid input because the algorithm negates all negative inputs and there is no positive representation of MIN_INT
	Example:
	`q8_24 x = Q24(0.005);`
	`q8_24 rad = dsp_math_atan(x);`

Type	`q8_24 dsp_math_atan(q8_24 x)`
Parameters	x input value Q8.24 format.
Returns	arctangent of x in radians between -pi/2 .. +pi/2

18 Scalar Math Functions: Exponential

Function	**dsp_math_exp**
Description	This function returns the natural exponent of a fixed point number. The input number has to be less than 4.8, otherwise the answer cannot be represented as a fixed point number. For input values larger than 3 there is a relatively large error in the last three bits of the answer.
Type	`q8_24 dsp_math_exp(q8_24 x)`
Parameters	x input value
Returns	e^x

19 Scalar Math Functions: Natural Logarithm

Function	**dsp_math_log**
Description	This function returns the natural logarithm (ln) of a fixed point number. The input number has to be positive.
Type	`q8_24 dsp_math_log(uq8_24 x)`
Parameters	x input value Q8.24 format.
Returns	ln(x).

20 Scalar Math Functions: Hyperbolic Sine

Function	**dsp_math_sinh**
Description	This function returns the hyperbolic sine (sinh) of a fixed point number. The input number has to be in the range [-5.5..5.5] in order to avoid overflow, and for values outside the [-4..4] range there are errors up to 4 bits in the result.
Type	`q8_24 dsp_math_sinh(q8_24 x)`
Parameters	x input value Q8.24 format.
Returns	sinh(x)

21 Scalar Math Functions: Hyperbolic Cosine

Function	**dsp_math_cosh**
Description	This function returns the hyperbolic cosine (cosh) of a fixed point number. The input number has to be in the range [-5.5..5.5] in order to avoid overflow, and for values outside the [-4..4] range there are errors up to 4 bits in the result.
Type	`q8_24 dsp_math_cosh(q8_24 x)`
Parameters	x input value Q8.24 format.
Returns	sinh(x)

22 Vector Math Functions: Minimum Value

Function	**dsp_vector_minimum**

Description	Vector Minimum.
	Locate the vector's first occurring minimum value, returning the index of the first occurring minimum value.
	Example:
	`int32_t samples[256];`
	`int32_t index = dsp_vector_minimum( samples, 256 );`

Type	`int32_t`
	`dsp_vector_minimum(const int32_t input_vector[],`
	`                   const int32_t vector_length)`

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.

23 Vector Math Functions: Maximum Value

Function	dsp_vector_maximum

Description	Vector Minimum.
	Locate the vector's first occurring maximum value, returning the index of the first occurring maximum value.
	Example:
	`int32_t samples[256];`
	`int32_t index = dsp_vector_maximum( samples, 256 );`

Type	`int32_t`
	`dsp_vector_maximum(const int32_t input_vector[],`
	`                   const int32_t 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.

24 Vector Math Functions: Element Negation

Function	**dsp_vector_negate**

Description	Vector negation: `R[i] = -X[i]`.
	This function sets each result element to the negative value of the corresponding input element.
	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:
	`int32_t samples[256] = { 0, 1, 2, 3, ... not shown for brevity };`
	`int32_t result[256];`
	`dsp_vector_negate( samples, result, 256 );`

Type	`void`
	`dsp_vector_negate(const int32_t input_vector_X[],`
	`                  int32_t result_vector_R[],`
	`                  const int32_t 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.

25 Vector Math Functions: Element Absolute Value

Function	**dsp_vector_abs**

Description	Vector absolute value: `R[i] = |X[i]|`.
	This function sets each element of the result vector to the absolute value of the corresponding input vector element.
	Example:
	`int32_t samples[256] = { 0, 1, 2, 3, ... not shown for brevity };`
	`int32_t result[256];`
	`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`
	`dsp_vector_abs(const int32_t input_vector_X[],`
	`               int32_t result_vector_R[],`
	`               const int32_t 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.

26 Vector Math Functions: Scalar Addition

Function	**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 scalar plus vector element result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.
	Example:
	`int32_t input_vector_X[256] = { 0, 1, 2, 3, ... not shown for brevity ↪ };`
	`int32_t input_scalar_A = Q28( 0.333 );`
	`int32_t result_vector_R[256];`
	`dsp_vector_adds( input_vector_X, scalar_value_A, result_vector_R, 256 ) ↪ ;`

Type	`void`
	`dsp_vector_adds(const int32_t input_vector_X[],`
	`int32_t input_scalar_A,`
	`int32_t result_vector_R[],`
	`const int32_t vector_length)`

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

27 Vector Math Functions: Scalar Multiplication

Function	**dsp_vector_muls**

Description	Vector / scalar multiplication: `R[i] = X[i] * A`.
	The elements in vector X are multiplied with the scalar A and stored in result vector R. Each multiplication produces a 64-bit result. If overflow occurs it is saturated at the minimum/maximum value given the fixed-point format and finally shifted right by `q_format` bits.
	Example:
	`int32_t input_vector_X[256] = { 0, 1, 2, 3, ... not shown for brevity ↪ };`
	`int32_t input_scalar_A = Q28( 0.333 );`
	`int32_t result_vector_R[256];`
	`dsp_vector_muls( input_vector_X, scalar_value_A, result_vector_R, 256 ) ↪ ;`

Type	`void`
	`dsp_vector_muls(const int32_t input_vector_X[],`
	`                int32_t input_scalar_A,`
	`                int32_t result_vector_R[],`
	`                const int32_t vector_length,`
	`                const int32_t q_format)`

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.

28 Vector Math Functions: Vector Addition

Function	**dsp_vector_addv**

Description	Vector / vector addition: `R[i] = X[i] + Y[i]`.
	32 bit addition is used to add Vector X to Vector Y. The results are stored in result vector R. They are not saturated so overflow may occur.
	Example:
	`int32_t input_vector_X[256] = { 0, 1, 2, 3, ... not shown for brevity ↪ };`
	`int32_t input_vector_Y[256] = { 0, -1, -2, -3, ... not shown for ↪ brevity };`
	`int32_t result_vector_R[256];`
	`dsp_vector_addv( input_vector_X, input_vector_Y, result_vector_R, 256 ) ↪ ;`

Type	`void`
	`dsp_vector_addv(const int32_t input_vector_X[],`
	`const int32_t input_vector_Y[],`
	`int32_t result_vector_R[],`
	`const int32_t 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.

29 Vector Math Functions: Vector Subtraction

Function	**dsp_vector_subv**

Description	Vector / vector subtraction: `R[i] = X[i] - Y[i]`.
	32 bit subtraction is used to subtract Vector Y from Vector X. The results are stored in result vector R. They are not saturated so overflow may occur.
	Example:
	`int32_t input_vector_X[256] = { 0, 1, 2, 3, ... not shown for brevity ↪ };`
	`int32_t input_vector_Y[256] = { 0, -1, -2, -3, ... not shown for ↪ brevity };`
	`int32_t result_vector_R[256];`
	`dsp_vector_subv( input_vector_X, input_vector_Y, result_vector_R, 256 ) ↪ ;`

Type	`void`
	`dsp_vector_subv(const int32_t input_vector_X[],`
	`const int32_t input_vector_Y[],`
	`int32_t result_vector_R[],`
	`const int32_t 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.

30 Vector Math Functions: Vector Multiplication

Function	**dsp_vector_mulv**

Description	Vector / vector multiplication: `R[i] = X[i] * Y[i]`.
	Each multiplication produces a 64-bit result. If overflow occurs it is saturated at the minimum/maximum value given the fixed-point format and finally shifted right by `q_format` bits.
	Example:
	`int32_t input_vector_X[256] = { 0, 1, 2, 3, ... not shown for brevity ↪ };`
	`int32_t input_vector_Y[256] = { 0, 1, 2, 3, ... not shown for brevity ↪ };`
	`int32_t result_vector_R[256];`
	`dsp_vector_mulv( input_vector_X, input_vector_Y, result_vector_R, 256, ↪ 28 );`

Type	`void`
	`dsp_vector_mulv(const int32_t input_vector_X[],`
	`const int32_t input_vector_Y[],`
	`int32_t result_vector_R[],`
	`const int32_t vector_length,`
	`const int32_t q_format)`

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).

31 Vector Math Functions: Vector multiplication and scalar addition

Function	dsp_vector_mulv_adds

Description	Vector multiplication and scalar addition: `R[i] = X[i] * Y[i] + A`.
	Each multiplication produces a 64-bit result. If overflow occurs it is saturated at the minimum/maximum value given the fixed-point format and finally shifted right by q_format bits. Then the scalar is added. Overflow may occur after the addition.
	Example:
	`int32_t input_vector_X[256] = { 0, 1, 2, 3, ... not shown for brevity };`
	`int32_t input_vector_Y[256] = { 0, 1, 2, 3, ... not shown for brevity };`
	`int32_t input_scalar_A = Q28( 0.333 );`
	`int32_t result_vector_R[256];`
	`dsp_vector_mulv_adds( input_vector_X, input_vector_Y, scalar_value_A,`
	`↪ result_vector_R, 256, 28 );`

Type	`void`
	`dsp_vector_mulv_adds(const int32_t input_vector_X[],`
	`const int32_t input_vector_Y[],`
	`int32_t input_scalar_A,`
	`int32_t result_vector_R[],`
	`const int32_t vector_length,`
	`const int32_t q_format)`

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).

32 Vector Math Functions: Scalar multiplication and vector addition

Function	dsp_vector_muls_addv

Description	Scalar multiplication and vector addition: R[i] = X[i] * A + Y[i].
	The elements in vector X are multiplied with the scalar A and then added to the corresponding element in Y.
	Each multiplication produces a 64-bit result. If overflow occurs it is saturated at the minimum/maximum value given the fixed-point format and finally shifted right by q_format bits. Overflow may occur after the addition Y[i].
	Example:
	`int32_t input_vector_X[256] = { 0, 1, 2, 3, ... not shown for brevity };`
	`int32_t input_scalar_A = Q28( 0.333 );`
	`int32_t input_vector_Y[256] = { 0, 1, 2, 3, ... not shown for brevity };`
	`int32_t result_vector_R[256];`
	`dsp_vector_muls_addv( input_vector_X, input_scalar_A, input_vector_Y,`
	`↪ result_vector_R, 256, 28 );`

Type	`void`
	`dsp_vector_muls_addv(const int32_t input_vector_X[],`
	`int32_t input_scalar_A,`
	`const int32_t input_vector_Y[],`
	`int32_t result_vector_R[],`
	`const int32_t vector_length,`
	`const int32_t q_format)`

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).

33 Vector Math Functions: Scalar multiplication and vector subtraction

Function	**dsp_vector_muls_subv**

Description	Scalar multiplication and vector subtraction: `R[i] = X[i] * A - Y[i]`.
	The elements in vector X are multiplied with the scalar A. The corresponding element in Y is subtracted from that.
	Each multiplication produces a 64-bit result. If overflow occurs it is saturated at the minimum/maximum value given the fixed-point format and finally shifted right by `q_format` bits. Overflow may occur after the subtraction of Y[i].
	Example:
	`int32_t input_vector_X[256];`
	`int32_t input_scalar_A = Q28( 0.333 );`
	`int32_t input_vector_Y[256];`
	`int32_t result_vector_R[256];`
	`dsp_vector_muls_subv( input_vector_X, input_scalar_A, input_vector_Y,`
	`↪ result_vector_R, 256, 28 );`

Type	`void`
	`dsp_vector_muls_subv(const int32_t input_vector_X[],`
	`int32_t input_scalar_A,`
	`const int32_t input_vector_Y[],`
	`int32_t result_vector_R[],`
	`const int32_t vector_length,`
	`const int32_t q_format)`

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).

34 Vector Math Functions: Vector multiplication and vector addition

Function	**dsp_vector_mulv_addv**

Description	Vector multiplication and vector addition: `R[i] = X[i] * Y[i] + Z[i]`.
	Each multiplication produces a 64-bit result. If overflow occurs it is saturated at the minimum/maximum value given the fixed-point format and finally shifted right by `q_format` bits. Overflow may occur after the addition of Z[i].
	Example:
	`int32_t input_vector_X[256];`
	`int32_t input_vector_Y[256];`
	`int32_t input_vector_Z[256];`
	`int32_t result_vector_R[256];`
	`dsp_vector_mulv_addv( input_vector_X, input_vector_Y, input_vector_Z,`
	`↪ result_vector_R, 256, 28 );`

Type	`void`
	`dsp_vector_mulv_addv(const int32_t input_vector_X[],`
	`const int32_t input_vector_Y[],`
	`const int32_t input_vector_Z[],`
	`int32_t result_vector_R[],`
	`const int32_t vector_length,`
	`const int32_t 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).

35 Vector Math Functions: Vector multiplication and vector subtraction

Function	**dsp_vector_mulv_subv**

Description	Vector multiplication and vector subtraction: `R[i] = X[i] * Y[i] - Z[i]`. Each multiplication produces a 64-bit result. If overflow occurs it is saturated at the minimum/maximum value given the fixed-point format and finally shifted right by `q_format` bits. Overflow may occur after the subtraction of Z[i].
	Example:
	`int32_t input_vector_X[256];`
	`int32_t input_vector_Y[256];`
	`int32_t input_vector_Z[256];`
	`int32_t result_vector_R[256];`
	`dsp_vector_mulv_subv( input_vector_X, input_vector_Y, input_vector_Z,`
	`↪ result_vector_R, 256, 28 );`

Type	`void`
	`dsp_vector_mulv_subv(const int32_t input_vector_X[],`
	`const int32_t input_vector_Y[],`
	`const int32_t input_vector_Z[],`
	`int32_t result_vector_R[],`
	`const int32_t vector_length,`
	`const int32_t 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).

36 Vector Math Functions: Complex vector multiplication

Function	dsp_vector_mulv_complex

Description	Complex vector / vector multiplication: `R[i] = X[i] * Y[i]`.
	Vectors X[i] and Y[i] are complex, with separate arrays for the real and imaginary components.
	Each multiplication produces a 64-bit result. If overflow occurs it is saturated at the minimum/maximum value given the fixed-point format and finally shifted right by `q_format` bits.
	Example:
	`int32_t input_vector_X_re[256] = { 0, 1, 2, 3, ... not shown for brevity };`
	`int32_t input_vector_X_im[256] = { 0, 1, 2, 3, ... not shown for brevity };`
	`int32_t input_vector_Y_re[256] = { 0, 1, 2, 3, ... not shown for brevity };`
	`int32_t input_vector_Y_im[256] = { 0, 1, 2, 3, ... not shown for brevity };`
	`int32_t result_vector_R_re[256];`
	`int32_t result_vector_R_im[256];`
	`dsp_vector_mulv_complex( input_vector_X_re, input_vector_X_im, input_vector_Y_re,`
	`↪ input_vector_Y_im, result_vector_R_re, result_vector_R_im, 256, 28 );`

Type	`void`
	`dsp_vector_mulv_complex(const int32_t input_vector_X_re[],`
	`const int32_t input_vector_X_im[],`
	`const int32_t input_vector_Y_re[],`
	`const int32_t input_vector_Y_im[],`
	`int32_t result_vector_R_re[],`
	`int32_t result_vector_R_im[],`
	`const int32_t vector_length,`
	`const int32_t q_format)`

Parameters	`input_vector_X_re`
	Pointer to real source data array X.
	
	`input_vector_X_im`
	Pointer to imaginary source data array X.
	
	`input_vector_Y_re`
	Pointer to real source data array Y.
	
	`input_vector_Y_im`
	Pointer to imaginary source data array Y.
	
	`result_vector_R_re`
	Pointer to the resulting real data array.
	
	`result_vector_R_im`
	Pointer to the resulting imaginary data array.
	
	`vector_length`
	Length of the input and output vectors.
	
	`q_format` Fixed point format (i.e. number of fractional bits).

37 Matrix Math Functions: Element Negation

Function	**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:
	`int32_t samples[8][32];`
	`int32_t result[8][32];`
	`dsp_matrix_negate( samples, result, 8, 32 );`

Type	`void`
	`dsp_matrix_negate(const int32_t input_matrix_X[],`
	`                  int32_t result_matrix_R[],`
	`                  const int32_t row_count,`
	`                  const int32_t column_count)`

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 and output matrices. `column_count` Number of columns in input and output matrices.

38 Matrix Math Functions: Scalar Addition

Function	**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:
	`int32_t input_matrix_X[8][32];`
	`int32_t input_scalar_A = Q28( 0.333 );`
	`int32_t result_vector_R[8][32];`
	`dsp_matrix_adds( input_matrix_X, scalar_matrix_A, result_matrix_R, 8,`
	`↪ 32 );`

Type	`void`
	`dsp_matrix_adds(const int32_t input_matrix_X[],`
	`int32_t input_scalar_A,`
	`int32_t result_matrix_R[],`
	`const int32_t row_count,`
	`const int32_t column_count)`

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.

39 Matrix Math Functions: Scalar Multiplication

Function	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 `dsp_math_multiply`).
	Example:
	`int32_t input_matrix_X[8][32];`
	`int32_t input_scalar_A = Q28( 0.333 );`
	`int32_t result_vector_R[8][32];`
	`dsp_matrix_muls( input_matrix_X, scalar_value_A, result_matrix_R, 256, 8, 32, 28 );`

Type	`void`
	`dsp_matrix_muls(const int32_t input_matrix_X[],`
	`                int32_t input_scalar_A,`
	`                int32_t result_matrix_R[],`
	`                const int32_t row_count,`
	`                const int32_t column_count,`
	`                const int32_t q_format)`

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).

40 Matrix Math Functions: Matrix Addition

Function	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:
	`int32_t input_matrix_X [256] = { 0, 1, 2, 3, ... not shown for brevity ↪ };`
	`int32_t input_matrix_Y [256] = { 0, 1, 2, 3, ... not shown for brevity ↪ };`
	`int32_t result_matrix_R[256];`
	`dsp_matrix_addm( input_matrix_X, input_matrix_Y, result_matrix_R, 8, 32 ↪ );`

Type	`void`
	`dsp_matrix_addm(const int32_t input_matrix_X[],`
	`const int32_t input_matrix_Y[],`
	`int32_t result_matrix_R[],`
	`const int32_t row_count,`
	`const int32_t column_count)`

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.

41 Matrix Math Functions: Matrix Subtraction

Function	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:
	`int32_t input_matrix_X [256] = { 0, 1, 2, 3, ... not shown for brevity ↪ };`
	`int32_t input_matrix_Y [256] = { 0, 1, 2, 3, ... not shown for brevity ↪ };`
	`int32_t result_matrix_R[256];`
	`dsp_matrix_subm( input_matrix_X, input_matrix_Y, result_matrix_R, 8, 32 ↪ );`

Type	`void`
	`dsp_matrix_subm(const int32_t input_matrix_X[],`
	`const int32_t input_matrix_Y[],`
	`int32_t result_matrix_R[],`
	`const int32_t row_count,`
	`const int32_t column_count)`

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.

42 Matrix Math Functions: Matrix Multiplication

Function	**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 `dsp_math_multiply`). The algorithm is optimised for performance using double word load and store instructions. As a result the matrices must have an even number of rows and columns.
	Example: MxN * NxP = MxP
	`int32_t input_matrix_X[rows_X][N];`
	`int32_t input_matrix_Y[columns_Y][N]; // transposed for better memory alighment !!`
	`int32_t result_matrix_R[rows_x][columns_Y];`
	`dsp_matrix_mulm( input_matrix_X, input_matrix_Y, result_matrix_R, 256, 8, 32, 28 );`
Type	`void`
	`dsp_matrix_mulm(const int32_t input_matrix_X[],`
	`   const int32_t input_matrix_Y[],`
	`   int32_t result_matrix_R[],`
	`   const int32_t rows_X,`
	`   const int32_t cols_Y,`
	`   const int32_t cols_X_rows_Y,`
	`   const int32_t 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.
	`rows_X` Number of rows in input matrix X. Must be even or will trap.
	`cols_Y` Number of columns input matrix Y. Must be even or will trap.
	`cols_X_rows_Y` Number of columns in input matrix X == rows in input matrix Y. Must be even or will trap.
	`q_format` Fixed point format (i.e. number of fractional bits).

43 Statistics Functions: Vector Absolute Sum

Function	dsp_vector_abs_sum

Description	Vector absolute sum: “R = (abs(X[0]) + abs(X[1]) + .
	This function computes the absolute sum of all vector elements
	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:
	`int32_t result;`
	`result = dsp_vector_abs_sum( input_vector, 256, 28 );`

Type	`int32_t`
	`dsp_vector_abs_sum(const int32_t *input_vector_X,`
	`                   const int32_t vector_length,`
	`                   const int32_t 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).

44 Statistics Functions: Vector Mean

Function	**dsp_vector_mean**

Description	Vector mean: “R = (X[0] + X[1] + .
	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:
	`int32_t result;`
	`result = dsp_vector_mean( input_vector, 256, 28 );`

Type	`int32_t`
	`dsp_vector_mean(const int32_t input_vector_X[],`
	`                const int32_t vector_length,`
	`                const int32_t 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).

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

Function	**dsp_vector_power**

Description	Vector power (sum of squares): “R = X[0]^2 + X[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 `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:
	`int32_t result;`
	`result = dsp_vector_power( input_vector, 256, 28 );`

Type	`int32_t`
	`dsp_vector_power(const int32_t input_vector_X[],`
	`                 const int32_t vector_length,`
	`                 const int32_t 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).

46 Statistics Functions: Root Mean Square (RMS)

Function	dsp_vector_rms

Description	Vector root mean square: "R = ((X[0]^2 + X[1]^2 + .
	This function computes the root-mean-square (RMS) of the values contained within the input vector.
	`result = 0`
	`for i = 0 to N-1: result += input_vector_X[i]`
	`return dsp_math_sqrt( result / vector_length )`
	Since each element in the vector is squared the behavior for fixed-point multiplication should be considered (see behavior for the function 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 dsp_math_sqrt; 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:
	`int32_t result;`
	`result = dsp_vector_rms( input_vector, 256, 28 );`

Type	`int32_t`
	`dsp_vector_rms(const int32_t input_vector_X[],`
	`               const int32_t vector_length,`
	`               const int32_t q_format)`

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).

47 Statistics Functions: Dot Product

Function	**dsp_vector_dotprod**

Description	Vector dot product: “R = X[0] * Y[0] + X[1] * Y[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 `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:
	`int32_t result;`
	`result = dsp_vector_dotprod( input_vector, 256, 28 );`

Type	`int32_t`
	`dsp_vector_dotprod(const int32_t input_vector_X[],`
	`    const int32_t input_vector_Y[],`
	`    const int32_t vector_length,`
	`    const int32_t 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).

48 Filter Design Functions: Notch Filter

Function	**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.
	`int32_t coeffs[5];`
	`dsp_design_biquad_notch( 0.25, 0.707, coeffs, 28 );`

Type	`void`
	`dsp_design_biquad_notch(double filter_frequency,`
	`double filter_Q,`
	`int32_t biquad_coeffs[5],`
	`const int32_t 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.
	
	`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: Low-pass Filter

Function	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.
	`int32_t coeffs[5];`
	`dsp_design_biquad_lowpass( 0.25, 0.707, coeffs, 28 );`

Type	`void`
	`dsp_design_biquad_lowpass(double filter_frequency,`
	`double filter_Q,`
	`int32_t biquad_coeffs[5],`
	`const int32_t 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).

50 Filter Design Functions: High-pass Filter

Function	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.
	`int32_t coeffs[5];`
	`dsp_design_biquad_highpass( 0.25, 0.707, coeffs, 28 );`

Type	`void`
	`dsp_design_biquad_highpass(double filter_frequency,`
	`                           double filter_Q,`
	`                           int32_t biquad_coeffs[5],`
	`                           const int32_t 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).

51 Filter Design Functions: All-pass Filter

Function	**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.
	`int32_t coeffs[5];`
	`dsp_design_biquad_allpass( 0.25, 0.707, coeffs, 28 );`

Type	`void`
	`dsp_design_biquad_allpass(double filter_frequency,`
	`double filter_Q,`
	`int32_t biquad_coeffs[5],`
	`const int32_t 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. `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).

52 Filter Design Functions: Band-pass Filter

Function	**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.
	`int32_t coeffs[5];`
	`dsp_design_biquad_bandpass( 0.20, 0.30, coeffs, 28 );`

Type	`void`
	`dsp_design_biquad_bandpass(double filter_frequency1,`
	`double filter_frequency2,`
	`int32_t biquad_coeffs[5],`
	`const int32_t q_format)`

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).

53 Filter Design Functions: Peaking Filter

Function	**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.
	`int32_t coeffs[5];`
	`dsp_design_biquad_peaking( 0.25, 0.707, 3.0, coeff[5], 28 );`

Type	`void`
	`dsp_design_biquad_peaking(double filter_frequency,`
	`double filter_Q,`
	`double peak_qain_db,`
	`int32_t biquad_coeffs[5],`
	`const int32_t 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).

54 Filter Design Functions: Bass Shelving Filter

Function	**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.
	`int32_t coeffs[5];`
	`dsp_design_biquad_lowshelf( 0.25, 0.707, +6.0, coeffs, 28 );`

Type	`void`
	`dsp_design_biquad_lowshelf(double filter_frequency,`
	`double filter_Q,`
	`double shelf_gain_db,`
	`int32_t biquad_coeffs[5],`
	`const int32_t 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).

55 Filter Design Functions: Treble Shelving Filter

Function	**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.
	`int32_t coeffs[5];`
	`dsp_design_biquad_highshelf( 0.25, 0.707, +6.0, coeffs, 28 );`

Type	`void`
	`dsp_design_biquad_highshelf(double filter_frequency,`
	`double filter_Q,`
	`double shelf_gain_db,`
	`int32_t biquad_coeffs[5],`
	`const int32_t 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).

56 FFT functions

Note: The method for processing two real signals with a single complex FFT was improved. It now requires only half the memory. The function dsp_fft_split_spectrum is used to split the combined N point output of dsp_fft_forward into two half-spectra of size N/2. One for each of the two real input signals. dsp_fft_merge_spectra is used to merge the two half-spectra into a combined spectrum that can be processed by dsp_fft_inverse.

Function	**dsp_fft_split_spectrum**
Description	This function splits the spectrum of the FFT of two real sequences. Takes the result of a double-packed dsp_complex_t array that has undergone an FFT. This function splits the result into two arrays, one for each real sequence, of length N/2. It is expected that the output will be cast by: dsp_complex_t (* restrict w)[2] = (dsp_complex_t (*)[2])pts; or a C equlivent. The 2 dimensional array w[2][N/2] can now be used to access the frequency information of the two real sequences independently, with the first index denoting the corresponding real sequence and the second index denoting the FFT frequency bin. Note that the DC component of the imaginary output spectrum (index zero) will contain the real component for the Nyquest rate.

Type	`void`
	`dsp_fft_split_spectrum(dsp_complex_t pts[], const uint32_t N)`

Parameters	`pts` Array of dsp_complex_t elements. `N` Number of points. Must be a power of two.

Function	**dsp_fft_merge_spectra**
Description	This function merges two split spectra. It is the exact inverse operation of dsp_fft_split_spectrum.

Type	`void`
	`dsp_fft_merge_spectra(dsp_complex_t pts[], const uint32_t N)`

Parameters	`pts` Array of dsp_complex_t elements. `N` Number of points. Must be a power of two.

Function	**dsp_fft_short_to_long**
Description	This function copies an array of dsp_complex_short_t elements to an array of an equal number of dsp_complex_t elements.

Continued on next page

Type	`void`
	`dsp_fft_short_to_long(const dsp_complex_short_t s[],`
	`                      dsp_complex_t l[],`
	`                      const uint32_t N)`

Parameters	`l` Array of dsp_complex_t elements. `s` Array of dsp_complex_short_t elements. `N` Number of points.

Function	**dsp_fft_long_to_short**
Description	This function copies an array of dsp_complex_t elements to an array of an equal number of dsp_complex_short_t elements.

Type	`void`
	`dsp_fft_long_to_short(const dsp_complex_t l[],`
	`                      dsp_complex_short_t s[],`
	`                      const uint32_t N)`

Parameters	`s` Array of dsp_complex_short_t elements. `l` Array of dsp_complex_t elements. `N` Number of points.

Function	**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 dsp_fft_bit_reverse() followed by dsp_fft_forward(), and for an inverse FFT it involves dsp_fft_bit_reverse() followed by dsp_fft_inverse(). In some cases bit reversal can be avoided, for example when computing a convolution.

Type	`void`
	`dsp_fft_bit_reverse(dsp_complex_t pts[], const uint32_t N)`

Parameters	`pts` Array of dsp_complex_t elements. `N` Number of points. Must be a power of two.

Function	dsp_fft_forward
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 magnitude of the FFT output is right shifted log2(N) times which corresponds to division by N as shown in EQUATION 31-5 of `http://www.dspguide.com/CH31.PDF`. 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 dsp_sine_N tables. The function does not perform bit reversed indexing of the input data. If required then dsp_fft_bit_reverse() should be called beforehand.

Type	`void dsp_fft_forward(dsp_complex_t pts[],`
	`                      const uint32_t N,`
	`                      const int32_t sine[])`

Parameters	`pts` Array of dsp_complex_t elements. `N` Number of points. Must be a power of two. `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 dsp_tables.c; for example, for a 1024 point FFT use dsp_sine_1024.

Function	dsp_fft_inverse
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. Max input range is -0x3fffffff..0x3fffffff. Integer overflow can occur with inputs outside of this range. 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 dsp_sine_N tables. The function does not perform bit reversed indexing of the input data. if required then dsp_fft_bit_reverse() should be called beforehand.

Type	`void dsp_fft_inverse(dsp_complex_t pts[],`
	`                      const uint32_t N,`
	`                      const int32_t sine[])`

Parameters	`pts` Array of dsp_complex_t elements. `N` Number of points. Must be a power of two. `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 dsp_tables.c; for example, for a 1024 point FFT use dsp_sine_1024.

57 Audio Sample Rate Conversion

The synchronous sample rate conversion functions in lib_dsp are now deprecated, and will be removed in a future release. Please use lib_src for all sample rate conversion functionality (synchronous and asynchronous), where it is now maintained.

The DSP library includes synchronous sample rate conversion functions to downsample (decimate) and oversample (upsample or interpolate) by a factor of three. In each case, the DSP processing is carried out each time a single output sample is required. In the case of the decimator, three input samples passed to filter with a resulting one sample output on calling the processing function. The interpolator produces an output sample each time the processing function is called but will require a single sample to be pushed into the filter every third cycle. All samples use Q31 format (left justified signed 32b integer).

Both sample rate converters are based on a 144 tap FIR filter with two sets of coefficients available, depending on application requirements:

- firos3_b_144.dat / firds3_b_144.dat - These filters have 20dB of attenuation at the nyquist frequency and a higher cutoff frequency
- firos3_144.dat / firds3_144.dat - These filters have 60dB of attenuation at the nyquist frequency but trade this off with a lower cutoff frequency

The filter coefficients may be selected by adjusting the line:

```
#define  FIROS3_COEFS_FILE
```

and:

```
#define  FIRDS3_COEFS_FILE
```

in the files dsp_os3.h (API for oversampling) and dsp_ds3.h (API for downsampling) respectively.

The OS3 processing takes up to 157 core cycles to compute a sample which translates to 1.57us at 100MHz or 2.512us at 62.5MHz core speed. This permits up to 8 channels of 16KHz -> 48KHz sample rate conversion in a single 62.5MHz core.

The DS3 processing takes up to 389 core cycles to compute a sample which translates to 3.89us at 100MHz or 6.224us at 62.5MHz core speed. This permits up to 9 channels of 48KHz -> 16KHz sample rate conversion in a single 62.5MHz core.

Both downsample and oversample functions return ERROR or NOERROR status codes as defined in return codes enums listed below.

The down sampling functions return the following error codes

```
FIRDS3_NO_ERROR
FIRDS3_ERROR
```

The up sampling functions return the following error codes

```
FIROS3_NO_ERROR
FIROS3_ERROR
```

For details on synchronous audio sample rate conversion by factors of two, or asynchronous audio sample rate conversion please see the XMOS Sample Rate Conversion Library[2].

57.1 DS3 Function API

[2] http://www.xmos.com/published/lib_src-userguide

Type	dsp_ds3_return_code_t
Description	Downsample by 3 return codes. This type describes the possible error status states from calls to the ds3 API.
Values	DSP_DS3_NO_ERROR DSP_DS3_ERROR

Function	dsp_ds3_init
Description	This function initialises the decimate by 3 function for a given instance.

Type	`dsp_ds3_return_code_t`
	`dsp_ds3_init(`
	`   dsp_ds3_ctrl_t *dsp_ds3_ctrl) __attribute__((deprecated("Please use _ds3_init'`

Parameters	`dsp_ds3_ctrl` DS3 control structure
Returns	DSP_DS3_NO_ERROR on success, DSP_DS3_ERROR on failure

Function	dsp_ds3_sync
Description	This function clears the decimate by 3 delay line for a given instance.

Type	`dsp_ds3_return_code_t`
	`dsp_ds3_sync(`
	`   dsp_ds3_ctrl_t *dsp_ds3_ctrl) __attribute__((deprecated("Please use _ds3_sync'`

Parameters	`dsp_ds3_ctrl` DS3 control structure
Returns	DSP_DS3_NO_ERROR on success, DSP_DS3_ERROR on failure

Function	dsp_ds3_proc
Description	This function performs the decimation on three input samples and outputs on sample The input and output buffers are pointed to by members of the dsp_ds3_ctrl structure.

Type	dsp_ds3_return_code_t
	`dsp_ds3_proc(`
	`   dsp_ds3_ctrl_t *dsp_ds3_ctrl) __attribute__((deprecated("Please use _ds3_proc'`

Parameters	`dsp_ds3_ctrl` DS3 control structure
Returns	DSP_DS3_NO_ERROR on success, DSP_DS3_ERROR on failure

57.2 OS3 Function API

Type	**dsp_os3_return_code_t**
Description	Oversample by 3 return codes. This type describes the possible error status states from calls to the os3 API.

Values	`DSP_OS3_NO_ERROR`
	
	`DSP_OS3_ERROR`

Function	**dsp_os3_init**
Description	This function initialises the oversample by 3 function for a given instance.

Type	dsp_os3_return_code_t
	`dsp_os3_init(`
	`   dsp_os3_ctrl_t *dsp_os3_ctrl) __attribute__((deprecated("Please use _os3_init'`

Parameters	`dsp_os3_ctrl` OS3 control structure
Returns	DSP_OS3_NO_ERROR on success, DSP_OS3_ERROR on failure

Function	**dsp_os3_sync**
Description	This function clears the oversample by 3 delay line for a given instance.

Type	dsp_os3_return_code_t
	`dsp_os3_sync(`
	`   dsp_os3_ctrl_t *dsp_os3_ctrl) __attribute__((deprecated("Please use _os3_sync'`

Parameters	`dsp_os3_ctrl` OS3 control structure
Returns	DSP_OS3_NO_ERROR on success, DSP_OS3_ERROR on failure

Function	**dsp_os3_input**
Description	This function pushes a single input sample into the filter It should be called three times for each FIROS3_proc call.

Type	`dsp_os3_return_code_t`
	`dsp_os3_input(`
	`   dsp_os3_ctrl_t *dsp_os3_ctrl) __attribute__((deprecated("Please use _os3_input`

Parameters	`dsp_os3_ctrl` OS3 control structure
Returns	DSP_OS3_NO_ERROR on success, DSP_OS3_ERROR on failure

Function	**dsp_os3_proc**
Description	This function performs the oversampling by 3 and outputs one sample The input and output buffers are pointed to by members of the dsp_os3_ctrl structure.

Type	`dsp_os3_return_code_t`
	`dsp_os3_proc(`
	`   dsp_os3_ctrl_t *dsp_os3_ctrl) __attribute__((deprecated("Please use _os3_proc"`

Parameters	`dsp_os3_ctrl` OS3 control structure
Returns	DSP_OS3_NO_ERROR on success, DSP_OS3_ERROR on failure

APPENDIX A - Known Issues

There are no known issues.

APPENDIX B - xCORE-200 DSP library change log

B.1 3.1.0

- Deprecated synchronous sample rate conversion functions - now maintained in lib_src
- Added functions to compute a fast fixed point atan2 and hypotenuse
- Added Q8 versions of the arc sine and arc cosine functions
- Added 16384 point sine table
- Improved performance of the forwards FFT function, with small reduction in memory footprint

B.2 3.0.0

- Added exponential and natural logarithm functions
- Added Hyperbolic sine and cosine
- Fixed Matrix Multiplication and improved performance
- Changed API prefix from `lib_dsp_` to dsp_.
- Changed lib_dsp_fft_complex_t to dsp_complex_t and lib_dsp_fft_complex_short_t to dsp_complex_short_t
- Various fixes in API documentation
- Added complex vector multiplication
- Added synchronous sample rate conversion (downsample or upsample by factor 3)

B.3 2.0.0

- FFT interface update. Consolidated interface and improved testing.
- Halved the memory for processing two real signals with a single complex FFT.
- Renamed *_transforms to *_fft to improve naming consistency
- Improved performance and accuracy of dsp_math_sqrt. Error is <= 1. Worst case performance is 96 cycles.
- int32_t and uint32_t now used more consistently.

B.4 1.0.4

- Added fixed point sine and cosine functions. Performance: 62 cycles for dsp_math_sin, 64 cycles for dsp_math_cos.
- Brute force testing of all input values proved accuracy to within one LSB (error is <= 1)
- Added short int complex and tworeals FFT and iFFT
- Improved Macros for converting from double to int and int to double.
- Added optimised fixed point atan function dsp_math_atan
- Most tests in math_app.xc are now self-checking. Improved error reporting.
- Option for performance measurements in 10ns cycles.

B.5 1.0.3

- Update to source code license and copyright

B.6 1.0.2

- FFT and inverse FFT for two complex short int signals

B.7 1.0.1

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

B.8 1.0.0

- Initial version