



Math Libraries Help

MPLAB Harmony Integrated Software Framework

Volume V: MPLAB Harmony Framework Reference

This volume provides API reference information for the framework libraries included in your installation of MPLAB Harmony.

Description



This volume is a programmer reference that details the interfaces to the libraries that comprise MPLAB Harmony and explains how to use the libraries individually to accomplish the tasks for which they were designed.

Math Libraries Help

This section provides descriptions of the Math libraries that are available in MPLAB Harmony.

DSP Fixed-Point Math Library

This topic describes the DSP Fixed-Point Math Library.

Introduction

The DSP Fixed-Point Library is available for the PIC32MZ family of microcontrollers. This library was created from optimized assembly routines written specifically for devices with microAptiv™ core features that utilize DSP ASE.

Description

The DSP Fixed-Point Library contains building block functions for developing digital signal processing algorithms. The library supports the Q15 and Q31 fractional data formats. The functions are implemented in efficient assembly specifically targeted at the DSP extensions in this core family. The library makes these functions available in a simple C-callable structure.

Functions included in the DSP Fixed-Point Library include complex math, vector math, matrix math, digital filters, and transforms. In many cases, these functions require specific data structures to operate, which are detailed in the header file and examples.

Using the Library

This topic describes the basic architecture of the DSP Fixed-Point Library and provides information and examples on its use.

Description

Interface Header File: [dsp.h](#), [libq.h](#)

The interface to the DSP Fixed-Point library is defined in the [dsp.h](#) header file. Any C language source (.c) file that uses the DSP Fixed-Point library must include [dsp.h](#). This file is automatically added to the project when 'use DSP library' is selected in MHC. In addition, this file name is added to the `#include` section of `system_definitions.h` after the configurations are completed through MHC.

Some functions make special use of the optimized fixed-point math library [libq.h](#). For use of those functions, the [libq.h](#) file must also be included in a project. The [libq.h](#) file is also installed with MPLAB Harmony. Specific notes within each function will describe if the function is dependent on the LibQ Fixed-Point Math Library.

Library Files: `dsp_pic32mz.a`, `LIBQ_Library.X.a`, and `dsp_pic32mz_ef.a`

The DSP Fixed-Point library archive (.a) files are installed with MPLAB Harmony. Although there are two PIC32 DSP files, the linker will only utilize one of these depending on your device usage. If you have a device using a FPU, the `dsp_pic32mz_ef.a` file is used by the linker. Otherwise, the linker will utilize `dsp_pic32mz.a`. In a future release of the tools, the linker in the MPLAB XC32 C/C++ Compiler may be changed to do this automatically. Both of these files are added to a project when the 'use DSP library' option is selected in MHC.

This library is only available in binary form, with prototypes for each function described in the [dsp.h](#) file.

For functions that are supported by the LibQ Fixed-Point Math library, the `LIBQ_Library.X.a` library must also be installed. This library is also available in binary form and is installed with MPLAB Harmony. In some cases, the linker in the MPLAB XC32 C/C++ Compiler may not acknowledge the LibQ Fixed-Point Math Library file. If the library is included in the project, and the message *undefined reference* is encountered upon compilation, it may be necessary to add the specific assembly file function to your project. The source code for each function can be found in the `<install-dir>/framework/math/libq` folder of your MPLAB Harmony installation.

Library Overview

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the DSP Fixed-Point Math Library.

Library Interface Section	Description
Complex Math Functions	General mathematical operations using a complex structure with the form (a + bi)
Vector Math Functions	Mathematical operations on a array of numbers or vector
Matrix Math Functions	Mathematical operations on a matrix
Digital Filter Functions	FIR and IIR filtering functions with various architectures
Transform Functions	FFT, Windows and related transform elements
Support Functions	Quick support functions for numerical transform

The DSP Fixed-Point Library uses fixed-point fractional functions to optimize execution speed. These functions limit the accuracy of the

calculations to the bits specified for the function. Due to parallelism in some operations, the 16-bit version of the functions are more efficient than their 32-bit counterparts. In many cases both 16-bit and 32-bit functions are available to give the user the choice of balance between speed and functional resolution.

Fractional representation of a real number is given by:

$Qn.m$

where:

- n is the number of data bits to the left of the radix point
 - m is the number of data bits to the right of the radix point
 - a signed bit is implied, and takes one bit of resolution
 - Shorthand may eliminate the leading 0, such as in Q0.15, which may be shortened to Q15, and similarly Q0.31, which is shortened to Q31
- $Qn.m$ numerical values are used by the library processing data as integers. In this format the n represents the number of integer bits, and the m represents the number of fractional bits. All values assume a sign bit in the most significant bit. Therefore, the range of the numerical value is: $-2^{(n-1)}$ to $[2^{(n-1)} - 2^{(-m)}]$; with a resolution of $2^{(-m)}$.

A Q16 format number (Q15.16) would range from -32768.0 (0x8000 0000) to 32767.99998474 with a precision of 0.000015259 (or 2^{-16}).

For example, a numerical representation of the number 3.14159 in Q2.13 notation would be:

$$3.14159 * 2^{13} = 25735.9 \Rightarrow 0x6488$$

And converting from the Q7.8 format with the value 0x1D89 would be:

$$0x1D89 / 2^8 = 7561 / 256 \Rightarrow 29.5316, \text{ accurate to } 0.00391$$

A majority of the DSP Fixed-Point Library uses functions with variables in Q15 or Q31 format. Representations of these numbers are given in **Data Types and Constants** in the [Library Interface](#) section, and generally are `int16_t` (for Q15 fractional representation) and `int32_t` (for Q31 fractional representation). This limits the equivalent numerical range to roughly -1.0 to 0.999999999. It is possible to represent other number ranges, but scaling before and after the function call are necessary.

All library functions will saturate the output if the value exceeds the maximum or is lower than the minimum allowable value for that resolution. Some prescaling may be necessary to prevent unwanted saturation in functions that may otherwise create calculation errors.








Table of Library Functions

Family	Function	Definition
Complex Math	Addition	DSP_ComplexAdd32
	Conjugate	DSP_ComplexConj32
	Dot Product	DSP_ComplexDotProd32
	Multiplication	DSP_ComplexMult32
	Scalar Multiplication	DSP_ComplexScalarMult32
	Subtraction	DSP_ComplexSub32
Digital Filter	FIR	DSP_FilterFIR32
	FIR Decimation	DSP_FilterFIRDecim32
	FIR Interpolation	DSP_FilterFIRInterp32
	FIR LMS	DSP_FilterLMS16
	IIR	DSP_FilterIIR16 ; DSP_FilterIIRSetup16
	IIR Biquad	DSP_FilterIIRBQ16 ; DSP_FilterIIRBQ16_fast ; DSP_FilterIIRBQ32
	IIR Biquad Cascade	DSP_FilterIIRBQ16_cascade8 ; DSP_FilterIIRBQ16_cascade8_fast
	IIR Biquad Parallel	DSP_FilterIIRBQ16_parallel8 ; DSP_FilterIIRBQ16_parallel8_fast
Matrix Math	Addition	DSP_MatrixAdd32
	Equality	DSP_MatrixEqual32
	Initialization	DSP_MatrixInit32
	Multiplication	DSP_MatrixMul32
	Scale	DSP_MatrixScale32
	Subtraction	DSP_MatrixSub32
Transform	Transpose	DSP_MatrixTranspose32
	FFT	DSP_TransformFFT16 ; DSP_TransformFFT32








	Setup factors	DSP_TransformFFT16_setup; DSP_TransformFFT32_setup
	inverse FFT	DSP_TransformIFFT16
	Windows	DSP_TransformWindow_Bart16; DSP_TransformWindow_Bart32; DSP_TransformWindow_Black16; DSP_TransformWindow_Black32; DSP_TransformWindow_Cosine16; DSP_TransformWindow_Cosine32; DSP_TransformWindow_Hamm16; DSP_TransformWindow_Hamm32; DSP_TransformWindow_Hann16; DSP_TransformWindow_Hann32; DSP_TransformWindow_Kaiser16; DSP_TransformWindow_Kaiser32;
	Window Initialization	DSP_TransformWinInit_Bart16; DSP_TransformWinInit_Bart32; DSP_TransformWinInit_Black16; DSP_TransformWinInit_Black32; DSP_TransformWinInit_Cosine16; DSP_TransformWinInit_Cosine32; DSP_TransformWinInit_Hamm16; DSP_TransformWinInit_Hamm32; DSP_TransformWinInit_Hann16; DSP_TransformWinInit_Hann32; DSP_TransformWinInit_Kaiser16; DSP_TransformWinInit_Kaiser32
Vector Math	Absolute value	DSP_VectorAbs16; DSP_VectorAbs32
	Addition	DSP_VectorAdd16; DSP_VectorAdd32
	Addition w/ constant	DSP_VectorAddc16; DSP_VectorAddc32
	Binary exponent	DSP_VectorBexp16; DSP_VectorBexp32
	Log, Exponent, Square root	DSP_VectorExp; DSP_VectorLog10; DSP_VectorLog2; DSP_VectorLn; DSP_VectorSqrt;
	Multiplication & Division	DSP_VectorMul16; DSP_VectorMul32; DSP_VectorDotp16; DSP_VectorDotp32; DSP_VectorMulc16; DSP_VectorMulc32; DSP_VectorDivC; DSP_VectorRecip
	Subtraction	DSP_VectorSub16; DSP_VectorSub32
	Power (sum of squares)	DSP_VectorSumSquares16; DSP_VectorSumSquares32
	Equality check	DSP_VectorChkEqu32
	Maximum	DSP_VectorMax32; DSP_VectorMaxIndex32
	Minimum	DSP_VectorMin32; DSP_VectorMinIndex32
	Copy & Fill	DSP_VectorCopy; DSP_VectorFill; DSP_VectorZeroPad
	Shift	DSP_VectorShift; DSP_VectorCopyReverse32
	Negate	DSP_VectorNegate
	Statistics	DSP_VectorAutocorr16; DSP_VectorMean32; DSP_VectorRMS16; DSP_VectorStdDev16; DSP_VectorVari16; DSP_VectorVariance







Library Interface

a) Complex Math Functions








	Name	Description
	DSP_ComplexAdd32	Calculates the sum of two complex numbers.
	DSP_ComplexConj16	Calculates the complex conjugate of a complex number.
	DSP_ComplexConj32	Calculates the complex conjugate of a complex number.
	DSP_ComplexDotProd32	Calculates the dot product of two complex numbers.
	DSP_ComplexMult32	Multiplies two complex numbers.
	DSP_ComplexScalarMult32	Multiplies a complex number and a scalar number.
	DSP_ComplexSub32	Calculates the difference of two complex numbers.

b) Digital Filter Functions






























	Name	Description
	DSP_FilterFIR32	Performs a Finite Infinite Response (FIR) filter on a vector.
	DSP_FilterFIRDecim32	Performs a decimating FIR filter on the input array.
	DSP_FilterFIRInterp32	Performs an interpolating FIR filter on the input array.
	DSP_FilterIIR16	Performs a single-sample cascaded biquad Infinite Impulse Response (IIR) filter.
	DSP_FilterIIRBQ16	Performs a single-pass IIR Biquad Filter.
	DSP_FilterIIRBQ16_cascade8	Performs a single-sample IIR Biquad Filter as a cascade of 8 series filters.
	DSP_FilterIIRBQ16_cascade8_fast	Performs a single-sample IIR Biquad Filter as a cascade of 8 series filters.

	DSP_FilterIIRBQ16_fast	Performs a single-pass IIR Biquad Filter.
	DSP_FilterIIRBQ16_parallel8	Performs a 8 parallel single-pass IIR Biquad Filters, and sums the result.
	DSP_FilterIIRBQ16_parallel8_fast	Performs a 8 parallel single-pass IIR Biquad Filters, and sums the result.
	DSP_FilterIIRBQ32	Performs a high resolution single-pass IIR Biquad Filter.
	DSP_FilterIIRSetup16	Converts biquad structure to coeffs array to set up IIR filter.
	DSP_FilterLMS16	Performs a single sample Least Mean Squares FIR Filter.





c) Matrix Math Functions

	Name	Description
	DSP_MatrixAdd32	Addition of two matrices $C = (A + B)$.
	DSP_MatrixEqual32	Equality of two matrices $C = (A)$.
	DSP_MatrixInit32	Initializes the first N elements of a Matrix to the value num.
	DSP_MatrixMul32	Multiplication of two matrices $C = A \times B$.
	DSP_MatrixScale32	Scales each element of an input buffer (matrix) by a fixed number.
	DSP_MatrixSub32	Subtraction of two matrices $C = (A - B)$.
	DSP_MatrixTranspose32	Transpose of a Matrix $C = A (T)$.

d) Transform Functions







	Name	Description
	DSP_TransformFFT16	Creates an Fast Fourier Transform (FFT) from a time domain input.
	DSP_TransformFFT16_setup	Creates FFT coefficients for use in the FFT16 function.
	DSP_TransformFFT32	Creates an Fast Fourier Transform (FFT) from a time domain input.
	DSP_TransformFFT32_setup	Creates FFT coefficients for use in the FFT32 function.
	DSP_TransformIFFT16	Creates an Inverse Fast Fourier Transform (FFT) from a frequency domain input.
	DSP_TransformWindow_Bart16	Perform a Bartlett window on a vector.
	DSP_TransformWindow_Bart32	Perform a Bartlett window on a vector.
	DSP_TransformWindow_Black16	Perform a Blackman window on a vector.
	DSP_TransformWindow_Black32	Perform a Blackman window on a vector.
	DSP_TransformWindow_Cosine16	Perform a Cosine (Sine) window on a vector.
	DSP_TransformWindow_Cosine32	Perform a Cosine (Sine) window on a vector.
	DSP_TransformWindow_Hamm16	Perform a Hamming window on a vector.
	DSP_TransformWindow_Hamm32	Perform a Hamming window on a vector.
	DSP_TransformWindow_Hann16	Perform a Hanning window on a vector.
	DSP_TransformWindow_Hann32	Perform a Hanning window on a vector.
	DSP_TransformWindow_Kaiser16	Perform a Kaiser window on a vector.
	DSP_TransformWindow_Kaiser32	Perform a Kaiser window on a vector.
	DSP_TransformWinInit_Bart16	Create a Bartlett window.
	DSP_TransformWinInit_Bart32	Create a Bartlett window.
	DSP_TransformWinInit_Black16	Create a Blackman window.
	DSP_TransformWinInit_Black32	Create a Blackman window.
	DSP_TransformWinInit_Cosine16	Create a Cosine (Sine) window.
	DSP_TransformWinInit_Cosine32	Create a Cosine (Sine) window.
	DSP_TransformWinInit_Hamm16	Create a Hamming window.
	DSP_TransformWinInit_Hamm32	Create a Hamming window.
	DSP_TransformWinInit_Hann16	Create a Hanning window.
	DSP_TransformWinInit_Hann32	Create a Hanning window.
	DSP_TransformWinInit_Kaiser16	Create a Kaiser window.
	DSP_TransformWinInit_Kaiser32	Create a Kaiser window.

e) Vector Math Functions

	Name	Description
	DSP_VectorAbs16	Calculate the absolute value of a vector.
	DSP_VectorAbs32	Calculate the absolute value of a vector.
	DSP_VectorAdd16	Calculate the sum of two vectors.
	DSP_VectorAdd32	Calculate the sum of two vectors.




	DSP_VectorAddc16	Calculate the sum of a vector and a constant.
	DSP_VectorAddc32	Calculate the sum of a vector and a constant.
	DSP_VectorAutocorr16	Computes the Autocorrelation of a Vector.
	DSP_VectorBexp16	Computes the maximum binary exponent of a vector.
	DSP_VectorBexp32	Computes the maximum binary exponent of a vector.
	DSP_VectorChkEqu32	Compares two input vectors, returns an integer '1' if equal, and '0' if not equal.
	DSP_VectorCopy	Copies the elements of one vector to another.
	DSP_VectorCopyReverse32	Reverses the order of elements in one vector and copies them into another.
	DSP_VectorDivC	Divides the first N elements of inVector by a constant divisor, and stores the result in outVector.
	DSP_VectorDotp16	Computes the dot product of two vectors, and scales the output by a binary factor.
	DSP_VectorDotp32	Computes the dot product of two vectors, and scales the output by a binary factor
	DSP_VectorExp	Computes the EXP (e^x) of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorFill	Fills an input vector with scalar data.
	DSP_VectorLn	Computes the Natural Log, $\ln(x)$, of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorLog10	Computes the $\log_{10}(x)$, of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorLog2	Computes the $\log_2(x)$ of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorMax32	Returns the maximum value of a vector.
	DSP_VectorMaxIndex32	Returns the index of the maximum value of a vector.
	DSP_VectorMean32	Calculates the mean average of an input vector.
	DSP_VectorMin32	Returns the minimum value of a vector.
	DSP_VectorMinIndex32	Returns the index of the minimum value of a vector.
	DSP_VectorMul16	Multiplication of a series of numbers in one vector to another vector.
	DSP_VectorMul32	Multiplication of a series of numbers in one vector to another vector.
	DSP_VectorMulc16	Multiplication of a series of numbers in one vector to a scalar value.
	DSP_VectorMulc32	Multiplication of a series of numbers in one vector to a scalar value.
	DSP_VectorNegate	Inverses the sign (negates) the elements of a vector.
	DSP_VectorRecip	Computes the reciprocal ($1/x$) of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorRMS16	Computes the root mean square (RMS) value of a vector.
	DSP_VectorShift	Shifts the data index of an input data vector.
	DSP_VectorSqrt	Computes the square root of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorStdDev16	Computes the Standard Deviation of a Vector.
	DSP_VectorSub16	Calculate the difference of two vectors.
	DSP_VectorSub32	Calculate the difference of two vectors.
	DSP_VectorSumSquares16	Computes the sum of squares of a vector, and scales the output by a binary factor.
	DSP_VectorSumSquares32	Computes the sum of squares of a vector, and scales the output by a binary factor.
	DSP_VectorVari16	Computes the variance of N elements of a Vector.
	DSP_VectorVariance	Computes the variance of N elements of inVector.
	DSP_VectorZeroPad	Fills an input vector with zeros.

f) Support Functions

	Name	Description
	mul16	multiply and shift integer
	mul16r	multiply and shift Q15
	mul32	multiply and shift Q31
	SAT16	saturate both positive and negative Q15
	SAT16N	saturate negative Q15
	SAT16P	saturate positive Q15

g) Data Types and Constants

	Name	Description
	biquad16	Q15 biquad
	int16c	Q15 complex number ($a + bi$)

	int32c	Q31 complex number (a + bi)
	matrix32	Q31 matrix
	PARM_EQUAL_FILTER	IIR BQ filter structure Q15 data, Q31 storage
	PARM_EQUAL_FILTER_16	IIR BQ filter structure Q15
	PARM_EQUAL_FILTER_32	IIR BQ filter structure Q31
	PARM_FILTER_GAIN	filter gain structure
	_PARM_EQUAL_FILTER	IIR BQ filter structure Q15 data, Q31 storage
	_PARM_EQUAL_FILTER_16	IIR BQ filter structure Q15
	_PARM_EQUAL_FILTER_32	IIR BQ filter structure Q31
	MAX16	maximum Q15
	MAX32	maximum Q31
	MIN16	minimum Q15
	MIN32	minimum Q31

Description

This section describes the Application Programming Interface (API) functions of the DSP Fixed-Point Library. Refer to each section for a detailed description.

a) Complex Math Functions

DSP_ComplexAdd32 Function

Calculates the sum of two complex numbers.

File

[dsp.h](#)

C

```
void DSP_ComplexAdd32(int32c * indata1, int32c * indata2, int32c * Output);
```

Returns

pointer to result complex numbers ([int32c](#))
None.

Description

Function DSP_ComplexAdd32:

```
void DSP_ComplexAdd32(int32c *indata1, int32c *indata2, int32c *Output);
```

Calculates the sum of two complex numbers, indata1 and indata2, and stores the complex result in Output. Complex numbers must be in the structural form that includes real and imaginary components. The function saturates the output values if maximum or minimum are exceeded. All values are in Q31 fractional data format. $(a + bi) + (c + di) \Rightarrow (a + c) + (b + d)i$

Remarks

None.

Preconditions

Complex numbers must be in the [int32c](#) format.

Example

```
int32c *res, result;
int32c *input1, *input2;
int32c test_complex_1 = {0x40000000,0x0CCCCCCC};
// (0.5 + 0.1i)
int32c test_complex_2 = {0x73333333,0xB3333334};
// (0.9 - 0.6i)
res=&result;
input1=&test_complex_1;
input2=&test_complex_2;

DSP_ComplexAdd32(input1, input2, res);
```



```
// result = {0x73333333, 0xC0000000} = (0.9 - 0.5i)
```

Parameters

Parameters	Description
indata1	pointer to input complex number (int32c)
indata2	pointer to input complex number (int32c)

DSP_ComplexConj16 Function

Calculates the complex conjugate of a complex number.

File

[dsp.h](#)

C

```
void DSP_ComplexConj16(int16c * indata, int16c * Output);
```

Returns

pointer to result complex numbers ([int16c](#))

None.

Description

DSP_ComplexConj16:

```
void DSP_ComplexConj16(int16c *indata, int16c *Output);
```

Calculates the complex conjugate of Indata, and stores the result in Outdata. Both numbers must be in the complex number data structure which includes real and imaginary components. Values are in Q15 fractional data format. The function will saturate the output if maximum or minimum values are exceeded. $(a + bi) \Rightarrow (a - bi)$

Remarks

None.

Preconditions

Complex numbers must be in the [int32c](#) format.

Example

```
int16c *res, result;
int16c *input1;
int16c test_complex_1 = {0x4000,0x0CCC};
//                               (0.5 + 0.1i)

res=&result;
input1=&test_complex_1;

DSP_ComplexConj16(input1, res);

// result = {0x4000, 0xF334} = (0.5 - 0.1i)
```

Parameters

Parameters	Description
indata	pointer to input complex number (int16c)

DSP_ComplexConj32 Function

Calculates the complex conjugate of a complex number.

File

[dsp.h](#)

C

```
void DSP_ComplexConj32(int32c * indata, int32c * Output);
```

Returns

pointer to result complex numbers ([int32c](#))
None.

Description

Function DSP_ComplexConj32:

```
void DSP_ComplexConj32(int32c *indata, int32c *Output);
```

Calculates the complex conjugate of indata, and stores the result in Output. Both numbers must be in the complex number data structure, which includes real and imaginary components. Values are in Q31 fractional data format. The function will saturate the output if maximum or minimum values are exceeded. $(a + bi) \Rightarrow (a - bi)$

Remarks

None.

Preconditions

Complex numbers must be in the [int32c](#) format.

Example

```
int32c *res, result;
int32c *input1;
int32c test_complex_1 = {0x40000000,0x0CCCCCCC};
//                               (0.5 + 0.1i)

res=&result;
input1=&test_complex_1;

DSP_ComplexConj32(input1, res);

// result = {0x40000000, 0xF3333334} = (0.5 - 0.1i)
```

Parameters

Parameters	Description
indata1	pointer to input complex number (int32c)

DSP_ComplexDotProd32 Function

Calculates the dot product of two complex numbers.

File

[dsp.h](#)

C

```
void DSP_ComplexDotProd32(int32c * indata1, int32c * indata2, int32c * Output);
```

Returns

pointer to result complex numbers ([int32c](#))
None.

Description

Function DSP_ComplexDotProd32:

```
void DSP_ComplexDotProd32(int32c *indata1, int32c *indata2, int32c *Output);
```

Calculates the dot product of two complex numbers, indata1 and indata2, and stores the result in Output. All numbers must be in complex structural format that includes real and imaginary components, and the numbers are in fractional Q31 format. The function will saturate the output if it exceeds maximum or minimum ratings. The formula for the dot product is as follows: $\text{Output}(\text{real}) = (\text{Input1.re} * \text{Input2.re}) + (\text{Input1.im} * \text{Input2.im})$; $\text{Output}(\text{img}) = [(\text{Input1.re} * \text{Input2.im}) - (\text{Input1.im} * \text{Input2.re})]i$ $(a + bi) \text{ dot } (c + di) \Rightarrow (a * c + b * d) + (a * d - b * c)i$

Remarks

None.

Preconditions

Complex numbers must be in the [int32c](#) format.

Example

```
int32c *res, result;
int32c *input1, *input2;
int32c test_complex_1 = {0x40000000,0x0CCCCCCC};
//          (0.5 + 0.1i)
int32c test_complex_2 = {0x73333333,0xB3333334};
//          (0.9 - 0.6i)
res=&result;
input1=&test_complex_1;
input2=&test_complex_2;

DSP_ComplexDotProd32(input1, input2, res);

// result = {0x31EB851E, 0xCE147AE3} = (0.39 - 0.39i)
```

Parameters

Parameters	Description
indata1	pointer to input complex number (int32c)
indata2	pointer to input complex number (int32c)

DSP_ComplexMult32 Function

Multiplies two complex numbers.

File

[dsp.h](#)

C

```
void DSP_ComplexMult32(int32c * indata1, int32c * indata2, int32c * Output);
```

Returns

pointer to result complex numbers ([int32c](#))

None.

Description

Function DSP_ComplexMult32:

```
void DSP_ComplexMult32(int32c *indata1, int32c *indata2, int32c *Output);
```

Multiplies two complex numbers, indata1 and indata2, and stores the complex result in Output. All numbers must be in the [int32c](#) complex data structure. All data is in Q31 fractional format. The function will saturate if maximum or minimum values are exceeded. Output(real) = (Input1.re * Input2.re) - (Input1.im * Input2.im); Output(img) = [(Input1.re * Input2.im) + (Input1.im * Input2.re)]i (a + bi) x (c + di) => (a * c - b * d) + (a * d + b * c)i

Remarks

None.

Preconditions

Complex numbers must be in the [int32c](#) format.

Example

```
int32c *res, result;
int32c *input1, *input2;
int32c test_complex_1 = {0x40000000,0x0CCCCCCC};
//          (0.5 + 0.1i)
int32c test_complex_2 = {0x73333333,0xB3333334};
//          (0.9 - 0.6i)
res=&result;
input1=&test_complex_1;
input2=&test_complex_2;

DSP_ComplexMult32(input1, input2, res);

// result = {0x4147AE14, 0xE51EB851} = (0.51 - 0.21i)
```

Parameters

Parameters	Description
indata1	pointer to input complex number (int32c)
indata2	pointer to input complex number (int32c)

DSP_ComplexScalarMult32 Function

Multiplies a complex number and a scalar number.

File

[dsp.h](#)

C

```
void DSP_ComplexScalarMult32(int32c * indata, int32_t scalar, int32c * Output);
```

Returns

pointer to result complex numbers ([int32c](#))

None.

Description

Function DSP_ComplexScalarMult32:

```
void DSP_ComplexScalarMult32(int32c *indata, int32_t scalar, int32c *Output);
```

Multiplies a complex number, indata, by a scalar number, scalar, and stores the result in Output. indata and Output must be in [int32c](#) structure with real and imaginary components. All data must be in the fractional Q31 format. The function will saturate if maximum or minimum values are exceeded. $Output(real) = (Input1.re * scalar)$; $Output(img) = [(Input1.im * scalar)]_j$ $(a + bi) * C \Rightarrow (a * C + b * Ci)$

Remarks

None.

Preconditions

Complex numbers must be in the [int32c](#) format.

Example

```
int32c *res, result;
int32c *input1;
int32_t scalarInput = 0x20000000; // 0.25
int32c test_complex_1 = {0x40000000, 0x0CCCCC};
// (0.5 + 0.1i)

res=&result;
input1=&test_complex_1;

DSP_ComplexScalarMult32(input1, scalarInput, res);

// result = {0x10000000, 0x03333333} = (0.125 + 0.025i)
```

Parameters

Parameters	Description
indata	pointer to input complex number (int32c)
Scalar	fractional scalar input value (int32_t)

DSP_ComplexSub32 Function

Calculates the difference of two complex numbers.

File

[dsp.h](#)

C

```
void DSP_ComplexSub32(int32c * indata1, int32c * indata2, int32c * Output);
```

Returns

pointer to result complex numbers ([int32c](#))
None.

Description

Function DSP_ComplexSub32:

```
void DSP_ComplexSub32(int32c *indata1, int32c *indata2, int32c *Output);
```

Calculates the difference of two complex numbers, indata1 less indata2, and stores the complex result in Output. Both numbers must be in a complex data structure, which includes real and imaginary components. The function saturates the output values if maximum or minimum are exceeded. Real and imaginary components are in the Q31 fractional data format. $(a + bi) - (c + di) \Rightarrow (a - c) + (b - d)i$

Remarks

None.

Preconditions

Complex numbers must be in the [int32c](#) format.

Example

```
int32c *res, result;
int32c *input1, *input2;
int32c test_complex_1 = {0x40000000,0x0CCCCCCC};
// (0.5 + 0.1i)
int32c test_complex_2 = {0x73333333,0xB3333334};
// (0.9 - 0.6i)
res=&result;
input1=&test_complex_1;
input2=&test_complex_2;

DSP_ComplexSub32(input1, input2, res);

// result = {0xCCCCCCD, 0x59999998} = (-0.4 + 0.7i)
```

Parameters

Parameters	Description
indata1	pointer to input complex number (int32c)
indata2	pointer to input complex number (int32c)

b) Digital Filter Functions

DSP_FilterFIR32 Function

Performs a Finite Infinite Response (FIR) filter on a vector.

File

[dsp.h](#)

C

```
void DSP_FilterFIR32(int32\_t * outdata, int32\_t * indata, int32\_t * coeffs2x, int32\_t * delayline, int N,
int K, int scale);
```

Returns

None.

Description

Function DSP_FilterFIR32:

```
void DSP_FilterFIR32(int32\_t *outdata, int32\_t *indata, int32\_t *coeffs2x, int32\_t *delayline, int N, int K, int scale);
```

Performs an FIR filter on the vector indata, and stores the output in the vector outdata. The number of samples processed in the array is given by N. The number of filter taps is given by K. The values are scaled upon input by the binary scaling factor (right shift), scale. The array of 2*K coefficients is contained in the array coeffs2x, where the values are in order b0, b1, b2... and repeated. Lastly the delayline is an array of K values that are initialized to zero and represent previous values. All values are in fractional Q31 data format. The function will saturate results if minimum

or maximum values are exceeded.

Remarks

Filter coefs must be repeated within the array. The array is twice as large as the number of taps, and the values are repeated in order b0, b1, b2,...bn, b0, b1, b2,... bn. The function updates the delayline array, which must be K elements long. The array should be initialized to zero prior to the first processing. It will contain values for processing cascaded filters within a loop.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four. K must be greater than 2 and a multiple of 2. delayline must have K elements, and be initialized to zero. coeffs2x must have 2*K elements.

Example

```
#define TAPS 4
#define numPOINTS 256

int filterN = numPOINTS;
int filterK = TAPS;
int filterScale = 1; // scale output by 1/2^1 => output * 0.5
int32_t FilterCoefs[TAPS*2] = {0x40000000, 0x20000000, 0x20000000, 0x20000000,
    0x40000000, 0x20000000, 0x20000000, 0x20000000};
// note repeated filter coefs, A B C D A B C D
//           0.5, 0.25, 0.25, 0.25, 0.5, 0.25, 0.25, 0.25

int32_t outFilterData[numPOINTS]={0};
int32_t inFilterData[numPOINTS];
int filterDelayLine[TAPS]={0};

while(true)
{
    // put some data into input array, inFilterData, here //

    DSP_FilterFIR32(outFilterData, inFilterData, FilterCoefs, filterDelayLine,
        filterN, filterK, filterScale);
}
```

Parameters

Parameters	Description
outdata	pointer to destination array of elements (int32_t)
indata	pointer to source array of elements (int32_t)
coeffs2x	pointer to an array of coefficients (int32_t)
delayline	pointer to an array of delay variables (int32_t)
N	number of points in the array to process (int) number of samples (int)
K	number of filter taps
scale	binary scaler divisor ($1 / 2^{\text{scale}}$) (int)

DSP_FilterFIRDecim32 Function

Performs a decimating FIR filter on the input array.

File

dsp.h

C

```
void DSP_FilterFIRDecim32(int32_t * outdata, int32_t * indata, int32_t * coeffs, int32_t * delayline, int
N, int K, int scale, int rate);
```

Returns

None.

Description

Function DSP_FilterFIRDecim32:

```
void DSP_FilterFIRDecim32(int32_t *outdata, int32_t *indata, int32_t *coeffs, int32_t *delayline, int N, int K, int scale, int rate);
```

Compute a FIR decimation filter on the input vector `indata`, and store the results to the vector `outdata`. The total number of output elements is set by `N`, and therefore the `outdata` array must be at least `N` in length. The decimation ratio is given by `rate`. The input is sampled every integer value of `rate`, skipping every $(rate-1)$ input samples. The input array must therefore be $(rate*N)$ samples long. The amount of filter taps is specified by `K`. `Coeffs` specifies the coefficients array. The `delayline` array holds delay inputs for calculation, and must be initialized to zero prior to calling the filter. Both `coeffs` and `delayline` must be `K` in length. `Scale` divides the input by a scaling factor by right shifting the number of bits $(1/2^{scale})$. All values of input, output, and `coeffs` are given in Q31 fractional data format. The function will saturate if the output value exceeds the maximum or minimum value.

$$Y = b_0 * X_0 + (b_1 * X(-1)) + (b_2 * X(-2))$$

Remarks

Coefs are loaded into the array with corresponding to the least delay first (`C0`, `C(-1)`, `C(-2)`). `K` must be greater than `rate`. Even while decimating the input stream, every input passes through the delayline. So FIR filters of arbitrary length will give the same output as a non-decimating FIR, just with fewer responses.

Preconditions

The pointers `outdata` and `indata` must be aligned on 4-byte boundaries. `delayline` must have `K` elements, and be initialized to zero. `coeffs` must have `K` elements. `outdata` must have `N` elements `indata` must have $(N*rate)$ elements

Example

```
#define N 8           // number of output samples
#define TAPS 5
#define SKIP 3

int testFilterN = N;           // number of output elements
int testFilterK = TAPS;       // number of taps
int testFilterRate = SKIP;    // decimation rate R
int32_t outFiltDataDec[N]={0};
int32_t *inTestFilter[N*SKIP];
int filtScaleNum = 1;         // scale output (1 / 2^n) => Y * 0.5

int32_t filtDelayTest[8]={0}; // always initialize to zero

// get pointer to input buffer here //
inTestFilter = &inputBuffer;

DSP_FilterFIRDecim32(outFiltDataDec, inTestFilter, inTestCoefs,
    filtDelayTest, testFilterN, testFilterK, filtScaleNum, testFilterRate);
```

Parameters

Parameters	Description
<code>outdata</code>	pointer to output array of elements (<code>int32_t</code>)
<code>indata</code>	pointer to input array of elements (<code>int32_t</code>)
<code>coeffs</code>	pointer to an array of coefficients (<code>int32_t</code>)
<code>delayline</code>	pointer to an array of delay variables (<code>int32_t</code>)
<code>N</code>	number of output elements to be processed (<code>int</code>)
<code>K</code>	number of filter taps and coeffs (<code>int</code>)
<code>scale</code>	binary scaler divisor $(1 / 2^{scale})$ (<code>int</code>)
<code>rate</code>	decimation ratio (<code>int</code>)

DSP_FilterFIRInterp32 Function

Performs an interpolating FIR filter on the input array.

File

[dsp.h](#)

C

```
void DSP_FilterFIRInterp32(int32_t * outdata, int32_t * indata, int32_t * coeffs, int32_t * delayline, int
N, int K, int scale, int rate);
```

Returns

None.

Description

Function DSP_FilterFIRInterp32:

```
void DSP_FilterFIRInterp32(int32_t *outdata, int32_t *indata, int32_t *coeffs, int32_t *delayline, int N, int K, int scale, int rate);
```

Perform an interpolating FIR filter on the first N samples of indata, and stores the result in outdata. The number of output elements is N*rate. The number of filter taps, K, must be an even multiple of N. The coefficients array, Coeffs, must be K elements long. The delay line array, delayline, must be K/R elements long, and be initialized to zero. All data elements must be in Q31 fractional data format. Scaling is performed via binary shift on the input equivalent to $(1/2^{\text{shift}})$. The function will saturate the output if it exceeds maximum or minimum values. The function creates R output values for each input value processed. The delayline of previous values is processed with R elements of the coefficient array. Numerically:

$$Y(1,0) = X(0)*C(0) + X(-1)*C(\text{rate}) + X(-2)*C(2*\text{rate}) \dots Y(1,1) = X(0)*C(1) + X(-1)*C(\text{rate}+1) + X(-2)*C(2*\text{rate} + 1) \dots Y(1,\text{rate}) = X(0)*C(N) + X(-1)*C(\text{rate}+N) + X(-2)*C(2*\text{rate} + N) \dots$$

where output Y corresponds to (input,rate) different outputs, input X has (M/rate) sample delays and C is the coefficient array.

Remarks

The function processes each input (rate) times. With each pass, coefficients are offset so that (K/rate) multiply accumulate cycles occur.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. delayline must have (K/R) elements, and be initialized to zero. K (taps) must be an even multiple of R (rate). outdata must have R*N elements.

Example

```
// interpret evenly 1/3 spaced values
#define N 4           // number of output samples
#define TAPS 6
#define INTERP 3

int ifiltN = N;
int ifiltK = TAPS; // k must be an even multiple of R
int ifiltR = INTERP;

int32_t ifiltOut[N*INTERP]={0};
int32_t ifiltDelay[2]={0}; // must be initialized to zero
int ifiltScale = 0; // no scaling

int32_t ifiltCoefsThirds[TAPS]={0x2AAAAAA9, 0x55555555,0x7FFFFFFE,
                                0x55555555,0x2AAAAAA9,0x00000000};
// 0.333333, 0.6666667, 0.9999999, 0.6666667, 0.33333333, 0

int32_t ifiltInput[N]={0x0CCCCCD, 0x19999999, 0x26666666, 0x33333333};
// 0.1, 0.2, 0.3, 0.4

DSP_FilterFIRInterp32(ifiltOut, ifiltInput, ifiltCoefsThirds, ifiltDelay,
                    ifiltN, ifiltK, ifiltScale, ifiltR);

// ifiltOut = {0x04444444, 0x08888889, 0x0CCCCCD, 0x11111111, 0x15555555, 0x19999999,
// 0x1DDDDDDD, 0x22222221, 0x26666665, 0x2AAAAAA,0x2EEEEEEE, 0x33333332}
// = 0.0333, 0.0667, 0.1, 0.1333, 0.1667,0.2, 0.2333, 0.2667, 0.3, 0.3333, 0.3667, 0.4
```

Parameters

Parameters	Description
outdata	pointer to output array of elements (int32_t)
indata	pointer to input array of elements (int32_t)
coeffs	pointer to an array of coefficients (int32_t)
delayline	pointer to an array of delay variables (int32_t)
N	number of output elements to be processed (int)
K	number of filter taps and coeffs (int)
scale	binary scaler divisor ($1 / 2^{\text{scale}}$) (int)
rate	decimation ratio (int)

DSP_FilterIIR16 Function

Performs a single-sample cascaded biquad Infinite Impulse Response (IIR) filter.

File

[dsp.h](#)

C

```
int16_t DSP_FilterIIR16(int16_t in, int16_t * coeffs, int16_t * delayline, int B, int scale);
```

Returns

Sample output Y (int16_t)

Description

Function DSP_FilterIIR16:

```
int16_t DSP_FilterIIR16(int16_t in, int16_t *coeffs, int16_t *delayline, int B, int scale);
```

Performs a single element cascaded biquad IIR filter on the input, in. The filter contains B number of biquad sections, and cascades the output of one to the input of the next. B must be greater than 2 and a multiple of 2. The int16_t output generated by the function is the computation from the final biquad stage. Delay pipeline array delayline must contain 2*B values and be initialized to zero prior to use. The coefficient array must contain 4*B elements, and must be set up in order of biquad a1, a2, b1, b2. A binary (right shift) factor, scale, will scale the output equivalent to (1/2^scale). All numerical values must be in Q15 fractional data format. The function will saturate values if maximum or minimum values are exceeded.

$$Y = X_0 + (b_1 * X(-1)) + (b_2 * X(-2)) + (a_1 * Y(-1)) + (a_2 * Y(-2))$$

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. B must be greater than 2 and a multiple of 2. delayline must have 2*B elements, and be initialized to zero. coeffs must have 4*B elements.

Example

```
#define B 8 // use * biquad filters in cascade

int dataSamples = 256;
int i, j;
biquad16 bquad[B];
int16_t coefs[4*B]= {0};
int16_t delaylines[2*B]= {0};
int16_t Y, X;
int scaleBquad = 1; // scale output (1 / 2^n) => Y * 0.5

// do something to set up coefs, for instance this example //
```

for (j=0; jRemarks:Filter coefs must be stored within the array as a1, a2, b1, b2, a1, a2, b1, b2, in order of biquads form input to output. A function to translate the coeffs from biquad structure to coeffs is available in DSP_FilterIIRSetup16. The function updates the delayline array, which must be 2*B elements **long**. The array should be initialized to zero prior to the first processing. It will contain values **for** processing cascaded filters within a loop.

Parameters

Parameters	Description
in	input data element X (int16_t)
coeffs	pointer to an array of coefficients (int16_t)
delayline	pointer to an array of delay variables (int16_t)
B	number of cascaded biquad filter groups to process (int)
scale	binary scaler divisor (1 / 2^scale) (int)

DSP_FilterIIRBQ16 Function

Performs a single-pass IIR Biquad Filter.

File

[dsp.h](#)

C

```
int16_t DSP_FilterIIRBQ16(int16_t Xin, PARM_EQUAL_FILTER * pFilter);
```

Returns

Sample output Y (int16_t)

Description

Function DSP_FilterIIRBQ16:

int16_t DSP_FilterIIRBQ16(int16_t Xin, PARM_EQUAL_FILTER *pFilter);

Calculates a single pass IIR biquad filter on Xin, and delivers the result as a 16-bit output. All math is performed using 32 bit instructions, with results truncated to 16-bits for the output. The delay register is stored as a 32-bit value for subsequent functions. All values are fractional Q15 and Q31, see data structure for specifics.

$$Y = X(0)*b_0 + (b_1 * X(-1)) + (b_2 * X(-2)) - (a_1 * Y(-1)) - (a_2 * Y(-2))$$

Remarks

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass. A gain of 2 has been hard coded into the function. This implies that all coefs should be input at half value. This is purposeful, since many filter designs need a div2 to have each coef between the required -1

Preconditions

Delay register values should be initialized to zero.

Example

```

PARM_EQUAL_FILTER *ptrFilterEQ;
PARM_EQUAL_FILTER FilterEQ;
uint16_t DataIn, DataOut;
ptrFilterEQ = &FilterEQ;

// 48KHz sampling; 1 KHz bandpass filter; Q=0.9
// divide by 2 and convert to Q15
// b0 = 0.06761171785499065
// b1 = 0
// b2 = -0.06761171785499065
// a1 = -1.848823142275648
// a2 = 0.8647765642900187

// note all coefs are half value of original design, gain handled in algorithm
ptrFiltEQ32->b[0]=0x0453; // feed forward b0 coef
ptrFiltEQ32->b[1]=0; // feed forward b1 coef
ptrFiltEQ32->b[2]=0xFBAD; // feed forward b2 coef

// note all coefs are half value of original design, gain handled in algorithm
// note subtract is handled in algorithm, so coefs go in at actual value
ptrFiltEQ32->a[0]=0x89AD; // feedback a1 coef
ptrFiltEQ32->a[1]=0x3758; // feedback a2 coef

for (i=0;i<256;i++)
{
// *** get some input data here
DataIn32 = three_hundred_hz[i];

DataOut = DSP_FilterIIRBQ16(DataIn, ptrFilterEQ);

// *** do something with the DataOut here
}

```

Parameters

Parameters	Description
Xin	input data element X (int16_t)
pFilter	pointer to filter coef and delay structure

DSP_FilterIIRBQ16_cascade8 Function

Performs a single-sample IIR Biquad Filter as a cascade of 8 series filters.

File

[dsp.h](#)

C

```
int16_t DSP_FilterIIRBQ16_cascade8(int16_t Xin, PARM_EQUAL_FILTER * pFilter_Array);
```

Returns

Sample output Y (int16_t)

Description

Function DSP_FilterIIRBQ16_cascade8:

```
int16_t DSP_FilterIIRBQ16_cascade8(int16_t Xin, PARM_EQUAL_FILTER *pFilter_Array);
```

Calculates a single pass IIR biquad cascade filter on Xin, and delivers the result as a 16-bit output. The cascade of filters is 8 unique biquad filters arranged in series such that the output of one is provided as the input to the next. A unique filter coefficient set is provided to each, and 32 bit delay lines are maintained for each. All math is performed using 32 bit instructions, which results truncated to 16-bits for the output. Global gain values are available on the output. Fracgain is a Q15 fractional gain value and expgain is a binary shift gain value. The combination of the two can be utilized to normalize the output as desired. All values are fractional Q15 and Q31, see data structure for specifics.

$Y = Y7 \leftarrow Y6 \leftarrow Y5 \leftarrow Y4 \leftarrow Y3 \leftarrow Y2 \leftarrow Y1 \leftarrow Y0$ where each Y_n filter element represents a unique IIR biquad: $Y_n = Y(n-1)*b_0 + (b_1 * Y(n-2)) + (b_2 * Y(n-3)) - (a_1 * Y(n-1)) - (a_2 * Y(n-2))$ and: for Y_0 ; $Y(n-1) = X_{in}(0)$

Remarks

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass. A gain of 2 has been hard coded into the function. This implies that all coefs should be input at half value. This is purposeful, since many filter designs need a div2 to have each coef between the required -1

Preconditions

Delay register values should be initialized to zero.

Example

```
PARM_EQUAL_FILTER filtArray[8];
uint16_t dataY, dataX;

// example to use 2 filter blocks as notch filters
// fill entire Filter Array with coefs
for (i=0;i<8;i++)
{
    filtArray[i].Z[0]=0;
    filtArray[i].Z[1]=0;

    // note all coefs are half value of original design, gain handled in algorithm
    // all pass
    filtArray[i].b[0]=0x4000;
    filtArray[i].b[1]=0; // feed forward b1 coef
    filtArray[i].b[2]=0; // feed forward b2 coef

    filtArray[i].a[0]=0; // feedback a1 coef
    filtArray[i].a[1]=0; // feedback a2 coef
}

// Unique filters for example
// 10KHz notch filter -- divide coefs by 2
b0 = 0.5883783602332997
b1 = -0.17124071441396285
b2 = 0.5883783602332997
a1 = -0.17124071441396285
a2 = 0.1767567204665992

// note all coefs are half value of original design, gain handled in algorithm
filtArray[3].b[0]=0x25a7; // feed forward b0 coef
filtArray[3].b[1]=0xf508; // feed forward b1 coef
filtArray[3].b[2]=0x25a7; // feed forward b2 coef

// note all coefs are half value of original design, gain handled in algorithm
// note subtract is handled in algorithm, so coefs go in at actual value
filtArray[3].a[0]=0xf508; // feedback a1 coef
filtArray[3].a[1]=0x0b4f; // feedback a2 coef
```

```

// 1 KHz notch filter -- divide coefs by 2
b0 = 0.9087554064944908
b1 = -1.7990948352036205
b2 = 0.9087554064944908
a1 = -1.7990948352036205
a2 = 0.8175108129889816

// note all coefs are half value of original design, gain handled in algorithm
filtArray[7].b[0]=0x3a29;           // feed forward b0 coef
filtArray[7].b[1]=0x8cdc;           // feed forward b1 coef
filtArray[7].b[2]=0x3a29;           // feed forward b2 coef

// note all coefs are half value of original design, gain handled in algorithm
// note subtract is handled in algorithm, so coefs go in at actual value
filtArray[7].a[0]=0x8cdc;           // feedback a1 coef
filtArray[7].a[1]=0x3452;           // feedback a2 coef

for (i=0;i<256;i++)
{
// *** get input data here
dataX = compound_300_1K_hz16[i];

dataY = DSP_FilterIIRBQ16_cascade8(dataX, filtArray);

// *** do something with the DataY here
}

```

Parameters

Parameters	Description
Xin	input data element X (int16_t)
pFilter	pointer to filter coef and delay structure

DSP_FilterIIRBQ16_cascade8_fast Function

Performs a single-sample IIR Biquad Filter as a cascade of 8 series filters.

File

[dsp.h](#)

C

```
int16_t DSP_FilterIIRBQ16_cascade8_fast(int16_t Xin, PARM_EQUAL_FILTER_16 * pFilter_Array);
```

Returns

Sample output Y (int16_t)

Description

Function DSP_FilterIIRBQ16_cascade8_fast:

```
int16_t DSP_FilterIIRBQ16_cascade8_fast(int16_t Xin, PARM_EQUAL_FILTER_16 *pFilter_Array);
```

Calculates a single pass IIR biquad cascade filter on Xin, and delivers the result as a 16-bit output. The cascade of filters is 8 unique biquad filters arranged in series such that the output of one is provided as the input to the next. A unique filter coefficient set is provided to each, and 16 bit delay lines are maintained for each. All math is performed using 16 bit instructions, which results rounded to 16-bits for the output. All values are fractional Q15, see data structure for specifics. The function will saturate the output should it exceed maximum or minimum values.

$Y = Y7 \leftarrow Y6 \leftarrow Y5 \leftarrow Y4 \leftarrow Y3 \leftarrow Y2 \leftarrow Y1 \leftarrow Y0$ where each Y_n filter element represents a unique IIR biquad: $Y_n = Y(n-1)*b_0 + (b_1 * Y(n-2)) + (b_2 * Y(n-3)) - (a_1 * Y(n-1)) - (a_2 * Y(n-2))$ and: for Y_0 ; $Y(n-1) = X_{in}(0)$

Remarks

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass. A gain of 2 has been hard coded into the function. This implies that all coefs should be input at half value. This is purposeful, since many filter designs need a div2 to have each coef between the required -1

Preconditions

Delay register values should be initialized to zero.

Example

```

PARAM_EQUAL_FILTER_16 filtArray[8];
uint16_t dataY, dataX;

// example to use 2 filter blocks as notch filters
// fill entire Filter Array with coeffs
for (i=0;i<8;i++)
{
    filtArray[i].Z[0]=0;
    filtArray[i].Z[1]=0;

    // note all coeffs are half value of original design, gain handled in algorithm
    // all pass
    filtArray[i].b[0]=0x4000;
    filtArray[i].b[1]=0; // feed forward b1 coef
    filtArray[i].b[2]=0; // feed forward b2 coef

    filtArray[i].a[0]=0; // feedback a1 coef
    filtArray[i].a[1]=0; // feedback a2 coef
}

// Unique filters for example
// 10KHz notch filter -- divide coeffs by 2
b0 = 0.5883783602332997
b1 = -0.17124071441396285
b2 = 0.5883783602332997
a1 = -0.17124071441396285
a2 = 0.1767567204665992

// note all coeffs are half value of original design, gain handled in algorithm
filtArray[3].b[0]=0x25a7; // feed forward b0 coef
filtArray[3].b[1]=0xf508; // feed forward b1 coef
filtArray[3].b[2]=0x25a7; // feed forward b2 coef

// note all coeffs are half value of original design, gain handled in algorithm
// note subtract is handled in algorithm, so coeffs go in at actual value
filtArray[3].a[0]=0xf508; // feedback a1 coef
filtArray[3].a[1]=0x0b4f; // feedback a2 coef

// 1 KHz notch filter -- divide coeffs by 2
b0 = 0.9087554064944908
b1 = -1.7990948352036205
b2 = 0.9087554064944908
a1 = -1.7990948352036205
a2 = 0.8175108129889816

// note all coeffs are half value of original design, gain handled in algorithm
filtArray[7].b[0]=0x3a29; // feed forward b0 coef
filtArray[7].b[1]=0x8cdc; // feed forward b1 coef
filtArray[7].b[2]=0x3a29; // feed forward b2 coef

// note all coeffs are half value of original design, gain handled in algorithm
// note subtract is handled in algorithm, so coeffs go in at actual value
filtArray[7].a[0]=0x8cdc; // feedback a1 coef
filtArray[7].a[1]=0x3452; // feedback a2 coef

for (i=0;i<256;i++)
{
    // *** get input data here
    dataX = compound_300_1K_hz16[i];

    dataY = DSP_FilterIIRBQ16_cascade8_fast(dataX, filtArray);

    // *** do something with the DataY here
}

```

Parameters

Parameters	Description
Xin	input data element X (int16_t)
pFilter	pointer to filter coef and delay structure

DSP_FilterIIRBQ16_fast Function

Performs a single-pass IIR Biquad Filter.

File

[dsp.h](#)

C

```
int16_t DSP_FilterIIRBQ16_fast(int16_t Xin, PARM_EQUAL_FILTER_16 * pFilter);
```

Returns

Sample output Y (int16_t)

Description

Function DSP_FilterIIRBQ16_fast:

```
int16_t DSP_FilterIIRBQ16_fast(int16_t Xin, PARM_EQUAL_FILTER_16 *pFilter);
```

Calculates a single pass IIR biquad filter on Xin, and delivers the result as a 16-bit output. All math is performed using 16 bit instructions, with results rounded to 16-bits for the output. The delay register is stored as a 16-bit value for subsequent functions. The function will saturate the results if maximum or minimum fractional values are exceeded. All values are fractional Q15 format.

$$Y = X(0)*b_0 + (b_1 * X(-1)) + (b_2 * X(-2)) - (a_1 * Y(-1)) - (a_2 * Y(-2))$$

Remarks

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass. A gain of 2 has been hard coded into the function. This implies that all coefs should be input at half value. This is purposeful, since many filter designs need a div2 to have each coef between the required -1

Preconditions

Delay register values should be initialized to zero.

Example

```
PARM_EQUAL_FILTER_16 *ptrFilterEQ;
PARM_EQUAL_FILTER_16 FilterEQ;
uint16_t DataIn, DataOut;
ptrFilterEQ = &FilterEQ;

// 48KHz sampling; 1 KHz bandpass filter; Q=0.9
// divide by 2 and convert to Q15
// b0 = 0.06761171785499065
// b1 = 0
// b2 = -0.06761171785499065
// a1 = -1.848823142275648
// a2 = 0.8647765642900187

// note all coefs are half value of original design, gain handled in algorithm
ptrFiltEQ32->b[0]=0x0453; // feed forward b0 coef
ptrFiltEQ32->b[1]=0; // feed forward b1 coef
ptrFiltEQ32->b[2]=0xFBAD; // feed forward b2 coef

// note all coefs are half value of original design, gain handled in algorithm
// note subtract is handled in algorithm, so coefs go in at actual value
ptrFiltEQ32->a[0]=0x89AD; // feedback a1 coef
ptrFiltEQ32->a[1]=0x3758; // feedback a2 coef

for (i=0;i<256;i++)
{
    // *** get some input data here
    DataIn32 = three_hundred_hz[i];
```

```

    DataOut = DSP_FilterIIRBQ16_fast(DataIn, ptrFilterEQ);

    // *** do something with the DataOut here
}

```

Parameters

Parameters	Description
Xin	input data element X (int16_t)
pFilter	pointer to filter coef and delay structure

DSP_FilterIIRBQ16_parallel8 Function

Performs a 8 parallel single-pass IIR Biquad Filters, and sums the result.

File

[dsp.h](#)

C

```
int16_t DSP_FilterIIRBQ16_parallel8(int16_t Xin, PARM_EQUAL_FILTER * pFilter);
```

Returns

Sample output Y (int16_t)

Description

Function DSP_FilterIIRBQ16_parallel8:

```
int16_t DSP_FilterIIRBQ16_parallel8(int16_t Xin, PARM_EQUAL_FILTER *pFilter);
```

Calculates a 8 parallel, single pass IIR biquad filters on Xin, sums the result and delivers the result as a 16-bit output. All math is performed using 32 bit instructions, which results truncated to 16-bits for the output. The delay register is stored as a 32-bit value for subsequent functions. Output is tuned by 2 multiplier factors. First each parallel section has a fractional gain (attenuation) that enables individual scaling of that section. Second, a global binary (log2N) gain is applied to the result. The combination of gain factors enable both gain and attenuation. All values are fractional Q15 and Q31, see data structure for specifics.

$Y = Y7/8 + Y6/8 + Y5/8 + Y4/8 + Y3/8 + Y2/8 + Y1/8 + Y0/8$ where each Y_n filter element represents a unique IIR biquad: $Y_n = X(0)*b_0 + (b_1 * X(n-1)) + (b_2 * X(n-2)) - (a_1 * Y_n(-1)) - (a_2 * Y_n(-2))$

Remarks

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass. A gain of 2 has been hard coded into the function. This implies that all coefs should be input at half value. This is purposeful, since many filter designs need a div2 to have each coef between the required -1

Preconditions

Delay register values should be initialized to zero. The sum of all fracgain should be ≤ 1

Example

```

PARM_EQUAL_FILTER filtArrayPara[8];
uint16_t dataY, dataX;

// fill entire Filter Array with coefs
for (i=0;i<8;i++)
{
    filtArrayPara[i].Z[0]=0;
    filtArrayPara[i].Z[1]=0;

    filtArrayPara[i].G.fracGain = 0x7FFF;           // gain = 1 default
    filtArrayPara[i].G.expGain = 1;                // == 2^N; gain of 2

    // note all coefs are half value of original design, gain handled in algorithm
    // none pass -- default
    filtArrayPara[i].b[0]=0;                       // feed forward b0 coef
    filtArrayPara[i].b[1]=0;                       // feed forward b1 coef
    filtArrayPara[i].b[2]=0;                       // feed forward b2 coef

    // note all coefs are half value of original design, gain handled in algorithm
    // note subtract is handled in algorithm, so coefs go in at actual value
    filtArrayPara[i].a[0]=0;                       // feedback a1 coef

```

```

    filtArrayPara[i].a[1]=0;          // feedback a2 coef
}

// 1K bandpass Q=0.9
filtArrayPara[7].G.fracGain = 0x4000; // gain = 0.5 because using 2 outputs
// note all coefs are half value of original design, gain handled in algorithm
filtArrayPara[7].b[0]=0x04ad;
filtArrayPara[7].b[1]=0;           // feed forward b1 coef
filtArrayPara[7].b[2]=0xfb53;     // feed forward b2 coef

// note all coefs are half value of original design, gain handled in algorithm
// note subtract is handled in algorithm, so coefs go in at actual value
filtArrayPara[7].a[0]=0x8a90;     // feedback a1 coef
filtArrayPara[7].a[1]=0x36a4;     // feedback a2 coef

// 300 Hz bandpass Q=0.9
filtArrayPara[6].G.fracGain = 0x1000; // gain = 0.125 as an example
// note all coefs are half value of original design, gain handled in algorithm
filtArrayPara[6].b[0]=0x017b;     // feed forward b0 coef
filtArrayPara[6].b[1]=0;           // feed forward b1 coef
filtArrayPara[6].b[2]=0xfe85;     // feed forward b2 coef

// note all coefs are half value of original design, gain handled in algorithm
// note subtract is handled in algorithm, so coefs go in at actual value
filtArrayPara[6].a[0]=0x8316;     // feedback a1 coef
filtArrayPara[6].a[1]=0x3d08;     // feedback a2 coef

for (i=0;i<256;i++)
{
    // *** get input data here
    dataX = compound_300_1K_hz16[i];

    dataY = DSP_FilterIIRBQ16_cascade8_fast(dataX, filtArray);

    // *** do something with the DataY here
}

```

Parameters

Parameters	Description
Xin	input data element X (int16_t)
pFilter	pointer to filter coef and delay structure

DSP_FilterIIRBQ16_parallel8_fast Function

Performs a 8 parallel single-pass IIR Biquad Filters, and sums the result.

File

[dsp.h](#)

C

```
int16_t DSP_FilterIIRBQ16_parallel8_fast(int16_t Xin, PARM_EQUAL_FILTER_16 * pFilter);
```

Returns

Sample output Y (int16_t)

Description

Function DSP_FilterIIRBQ16_parallel8_fast:

```
int16_t DSP_FilterIIRBQ16_parallel8_fast(int16_t Xin, PARM_EQUAL_FILTER_16 *pFilter);
```

Calculates a 8 parallel, single pass IIR biquad filters on Xin, sums the result and delivers the result as a 16-bit output. All math is performed using 16 bit instructions, which results rounded to 16-bits for the output. The delay register is stored as a 16-bit value for subsequent functions. Output is tuned by 2 multiplier factors. First each parallel section has a fractional gain (attenuation) that enables individual scaling of that section. Second, a global binary (log2N) gain is applied to the result. The combination of gain factors enable both gain and attenuation. All values are fractional Q15. The function will round outputs and saturate if maximum or minimum values are exceeded.

$Y = Y7/8 + Y6/8 + Y5/8 + Y4/8 + Y3/8 + Y2/8 + Y1/8 + Y0/8$ where each Y_n filter element represents a unique IIR biquad: $Y_n = X(0)*b_0 + (b_1 * X(n-1)) + (b_2 * X(n-2)) - (a_1 * Y_n(-1)) - (a_2 * Y_n(-2))$

Remarks

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass. A gain of 2 has been hard coded into the function. This implies that all coefs should be input at half value. This is purposeful, since many filter designs need a div2 to have each coef between the required -1

Preconditions

Delay register values should be initialized to zero. The sum of all fracgain should be ≤ 1

Example

```

PARAM_EQUAL_FILTER_16 filtArrayPara[8];    // note change in data structure
uint16_t dataY, dataX;

// fill entire Filter Array with coefs
for (i=0;i<8;i++)
{
    filtArrayPara[i].Z[0]=0;
    filtArrayPara[i].Z[1]=0;

    filtArrayPara[i].G.fracGain = 0x7FFF;    // gain = 1 default
    filtArrayPara[i].G.expGain = 1;        // log2N; gain of 2

    // note all coefs are half value of original design, gain handled in algorithm
    // none pass -- default
    filtArrayPara[i].b[0]=0;                // feed forward b0 coef
    filtArrayPara[i].b[1]=0;                // feed forward b1 coef
    filtArrayPara[i].b[2]=0;                // feed forward b2 coef

    // note all coefs are half value of original design, gain handled in algorithm
    // note subtract is handled in algorithm, so coefs go in at actual value
    filtArrayPara[i].a[0]=0;                // feedback a1 coef
    filtArrayPara[i].a[1]=0;                // feedback a2 coef
}

// 1K bandpass Q=0.9
filtArrayPara[7].G.fracGain = 0x4000;    // gain = 0.5 because using 2 outputs
// note all coefs are half value of original design, gain handled in algorithm
filtArrayPara[7].b[0]=0x04ad;
filtArrayPara[7].b[1]=0;                // feed forward b1 coef
filtArrayPara[7].b[2]=0xfb53;            // feed forward b2 coef

// note all coefs are half value of original design, gain handled in algorithm
// note subtract is handled in algorithm, so coefs go in at actual value
filtArrayPara[7].a[0]=0x8a90;            // feedback a1 coef
filtArrayPara[7].a[1]=0x36a4;            // feedback a2 coef

// 300 Hz bandpass Q=0.9
filtArrayPara[6].G.fracGain = 0x1000;    // gain = 0.125 as an example
// note all coefs are half value of original design, gain handled in algorithm
filtArrayPara[6].b[0]=0x017b;            // feed forward b0 coef
filtArrayPara[6].b[1]=0;                // feed forward b1 coef
filtArrayPara[6].b[2]=0xfe85;            // feed forward b2 coef

// note all coefs are half value of original design, gain handled in algorithm
// note subtract is handled in algorithm, so coefs go in at actual value
filtArrayPara[6].a[0]=0x8316;            // feedback a1 coef
filtArrayPara[6].a[1]=0x3d08;            // feedback a2 coef

for (i=0;i<256;i++)
{
    // *** get input data here
    dataX = compound_300_1K_hz16[i];

    dataY = DSP_FilterIIRBQ16_cascade8_fast(dataX, filtArray);

    // *** do something with the DataY here
}

```

Parameters

Parameters	Description
Xin	input data element X (int16_t)
pFilter	pointer to filter coef and delay structure

DSP_FilterIIRBQ32 Function

Performs a high resolution single-pass IIR Biquad Filter.

File

[dsp.h](#)

C

```
int32_t DSP_FilterIIRBQ32(int32_t Xin, PARM_EQUAL_FILTER_32 * pFilter);
```

Returns

Sample output Y (int32_t)

Description

Function DSP_FilterIIRBQ32:

```
int32_t DSP_FilterIIRBQ32(int32_t Xin, PARM_EQUAL_FILTER_32 *pFilter);
```

Calculates a single pass IIR biquad filter on Xin, and delivers the result as a 16-bit output. All math is performed using 32 bit instructions, with results truncated to 32-bits for the output. The delay register is stored as a 32-bit value for subsequent functions. All values are fractional Q31, see data structure for specifics.

$$Y = X(0)*b_0 + (b_1 * X(-1)) + (b_2 * X(-2)) - (a_1 * Y(-1)) - (a_2 * Y(-2))$$

Remarks

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass. A gain of 2 has been hard coded into the function. This implies that all coefs should be input at half value. This is purposeful, since many filter designs need a div2 to have each coef between the required -1

Preconditions

Delay register values should be initialized to zero.

Example

```
PARM_EQUAL_FILTER_32 *ptrFiltEQ32;
PARM_EQUAL_FILTER_32 FilterEQ32;
int32_t DataIn32, DataOut32;

ptrFiltEQ32 = &FilterEQ32;

ptrFiltEQ32->Z[0]=0;
ptrFiltEQ32->Z[1]=0;

// 1000 Hz Q= 0.9 BP filter design, 44.1K sampling
//
// b0 = 0.07311778239751009 forward
// b1 = 0
// b2 = -0.07311778239751009
// a1 = -1.8349811166056893 back
// a2 = 0.8537644352049799

// note all coefs are half value of original design, gain handled in algorithm
ptrFiltEQ32->b[0]=0x04ADF635; // feed forward b0 coef
ptrFiltEQ32->b[1]=0; // feed forward b1 coef
ptrFiltEQ32->b[2]=0xFB5209CB; // feed forward b2 coef

// note all coefs are half value of original design, gain handled in algorithm
// note subtract is handled in algorithm, so coefs go in at actual value
ptrFiltEQ32->a[0]=0x8A8FAB5D; // feedback a1 coef
ptrFiltEQ32->a[1]=0x36A41395; // feedback a2 coef

for (i=0;i<256;i++)
```

```

{
  // *** get input data here
  DataIn32 = three_hundred_hz[i];

  DataOut32 = DSP_FilterIIRBQ32(DataIn32, ptrFiltEQ32);

  // *** do something with the DataOut32 here
}

```

Parameters

Parameters	Description
Xin	input data element X (int32_t)
pFilter	pointer to high resolution filter coef and delay structure

DSP_FilterIIRSetup16 Function

Converts biquad structure to coeffs array to set up IIR filter.

File

[dsp.h](#)

C

```
void DSP_FilterIIRSetup16(int16_t * coeffs, biquad16 * bq, int B);
```

Returns

None.

Description

Function DSP_FilterIIRSetup16:

```
void DSP_FilterIIRSetup16(int16_t *coeffs, biquad16 *bq, int B);
```

Converts an array of biquad coefficients, bq, into an linear array of coefficients, coeffs. The output array must be 4*B elements long. The number of biquads in the resulting factor is given by B. All numerical values must be in Q15 fractional data format.

Remarks

None.

Preconditions

coeffs must have 4*B elements.

Example

see DSP_FilterIIR16 [for](#) example.

Parameters

Parameters	Description
coeffs	pointer to an array of coefficients (int16_t)
bq	pointer to array of biquad structure filter coefs (biquad16)
B	number of cascaded biquad filter groups to process (int)

DSP_FilterLMS16 Function

Performs a single sample Least Mean Squares FIR Filter.

File

[dsp.h](#)

C

```
int16_t DSP_FilterLMS16(int16_t in, int16_t ref, int16_t * coeffs, int16_t * delayline, int16_t * error,
int K, int16_t mu);
```

Returns

(int16_t) - FIR filter output

Description

Function DSP_FilterLMS16:

```
int16_t DSP_FilterLMS16(int16_t in, int16_t ref, int16_t *coeffs, int16_t *delayline, int16_t *error, int K, int16_t mu);
```

Computes an LMS adaptive filter on the input in. Filter output of the FIR is given as a 16 bit value. The filter target is ref, and the calculation difference between the output and the target is error. The filter adapts its coefficients, coefs, on each pass. The number of coefficients (filter taps) is given by the value K. The delayline array should be initialized to zero prior to calling the filter for the first time, and have K elements. The value mu is the rate at which the filter adapts. All values are Q15 fractional numbers. The function will saturate the output if it exceeds maximum or minimum values. The LMS will adapt its coefs to attempt to drive the output value toward the ref value. The rate of adaption on each pass depends on mu and the error from the previous calculation.

Remarks

Filter coefs may start at random or zero value, but convergence is dependent on the amount of update required.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. delayline must have (K) elements, and be initialized to zero. K (taps) must be a multiple of 4, and >= 8. mu must be positive.

Example

```
#define lmsTAPS 8

int16_t lmsOut;
int lmsTaps = lmsTAPS;

int16_t lmsCoefs[lmsTAPS]={0x5000, 0x4000,0x3000, 0x2000,0x1000, 0x0000,0xF000, 0xE000};
int16_t lmsDelay[lmsTAPS]={0};
int16_t *ptrLMSError;
int16_t lmsError = 0x0200;
int16_t inVal=0;

int16_t refVal = 0x0CCC; // some target value = 0.1
int16_t lmsAdapt = 0x3000;

ptrLMSError = &lmsError;

for (i=0;i<200;i++)
{
// get some input value here //
if (i < 100)
{
inVal = 0x4233;
}
else
{
inVal = 0xCF10;
}

lmsOut = DSP_FilterLMS16(inVal, refVal, lmsCoefs, lmsDelay,
ptrLMSError, lmsTaps, lmsAdapt);
}
```

Parameters

Parameters	Description
in	input data value (int16_t)
ref	target output value (int16_t)
coeffs	pointer to an array of coefficients (int16_t)
delayline	pointer to an array of delay variables (int16_t)
error	output minus reference (int16_t)
K	number of filter taps and coeffs (int)
mu	adaption rate (int16_t)

c) Matrix Math Functions

DSP_MatrixAdd32 Function

Addition of two matrices $C = (A + B)$.

File

[dsp.h](#)

C

```
void DSP_MatrixAdd32(matrix32 * resMat, matrix32 * srcMat1, matrix32 * srcMat2);
```

Returns

None.

Description

Function DSP_MatrixAdd32:

```
void DSP_MatrixAdd32(matrix32 *resMat, matrix32 *srcMat1, matrix32 *srcMat2);
```

Vector summation of two matrices, where both have 32-bit integer elements. The resulting output will saturate if element addition exceeds [MAX32](#) or [MIN32](#).

Remarks

Execution Time (cycles): 225 cycles + 23 / matrix_element. The function will saturate the output value if it exceeds maximum limits per element.

Preconditions

Both matrices must be equivalent in rows and columns. Both Matrices must be set into structure (ROWS, COLUMNS, vector_pointer).

Example

```
#define ROW 2
#define COL 2

matrix32 *resMat, *srcMat1, *srcMat2;
int32_t result[ROW*COL];

int32_t matA[ROW*COL] = {1,2,3,4};
int32_t matB[ROW*COL] = {2,4,6,8};

matrix32 mat, mat2, mat3;
resMat=&mat;
srcMat1=&mat2;
srcMat2=&mat3;

srcMat1->row=ROW;
srcMat1->col=COL;
srcMat1->pMatrix=matA;

srcMat2->col=COL;
srcMat2->row=ROW;
srcMat2->pMatrix=matB;

resMat->row=ROW;
resMat->col=COL;
resMat->pMatrix=result;

DSP_MatrixAdd32(resMat, srcMat1, srcMat2);

// result[i] = matA[i] + matB[i] = {3,6,9,0xA}
```

Parameters

Parameters	Description
resMat	pointer to new sum Matrix C (*int32_t)
srcMat1	pointer to the Matrix A structure (*int32_t)
srcMat2	pointer to the Matrix B structure (*int32_t)

DSP_MatrixEqual32 Function

Equality of two matrices $C = (A)$.

File

[dsp.h](#)

C

```
void DSP_MatrixEqual32(matrix32 * resMat, matrix32 * srcMat);
```

Returns

None.

Description

Function DSP_MatrixEqual32:

```
void DSP_MatrixEqual32(matrix32 *resMat, matrix32 *srcMat);
```

Vector copy of all elements from one matrix to another. C is a duplicate of A.

Remarks

Execution Time (cycles): 163 cycles + 12 / matrix_element.

Preconditions

None.

Example

```
#define ROW 2
#define COL 2

matrix32 *resMat, *srcMat1, *srcMat2;
int32_t result[ROW*COL];

int32_t matA[ROW*COL] = {5,2,-3,8};

matrix32 mat, mat2;
resMat=&mat;
srcMat1=&mat2;

srcMat1->row=ROW;
srcMat1->col=COL;
srcMat1->pMatrix=matA;

resMat->row=ROW;
resMat->col=COL;
resMat->pMatrix=result;

DSP_MatrixEqual32(resMat, srcMat1, srcMat2);

// result[i] = matA[i] = {5, 2, -3, 8}
```

Parameters

Parameters	Description
resMat	pointer to completed new Matrix C (*int32_t)
srcMat	pointer to the Matrix A structure (*int32_t)

DSP_MatrixInit32 Function

Initializes the first N elements of a Matrix to the value num.

File

[dsp.h](#)

C

```
void DSP_MatrixInit32(int32_t * data_buffer, int N, int32_t num);
```

Returns

None.

Description

Function DSP_MatrixInit32:

```
void DSP_MatrixInit32(int32_t *data_buffer, int N, int32_t num);
```

Copy the value num into the first N Matrix elements of data_buffer.

Remarks

None.

Preconditions

data_buffer must be predefined to be equal to or greater than N elements. N must be a factor of four, or it will truncate to the nearest factor of four.

Example

```
#define ROW 3
#define COL 3

int32_t numElements = 4; // multiple of 4
int valueElements = -1;

int32_t matA[ROW*COL] = {5,2,-3,8,4,2,-6,8,9};

DSP_MatrixInit32(matA, numElements, valueElements);

// matA[i] = {-1,-1,-1,-1,4,2,-6,8,9}
```

Parameters

Parameters	Description
data_buffer	pointer to the Matrix to be initialized (int32_t[M*N])
N	number of elements to be initialized (int32_t)
num	value to be initialized into the matrix (int32_t)

DSP_MatrixMul32 Function

Multiplication of two matrices $C = A \times B$.

File

[dsp.h](#)

C

```
void DSP_MatrixMul32(matrix32 * resMat, matrix32 * srcMat1, matrix32 * srcMat2);
```

Returns

None.

Description

Function DSP_MatrixMul32:

```
void DSP_MatrixMul32(matrix32 *resMat, matrix32 *srcMat1, matrix32 *srcMat2);
```

Multiplication of two matrices, with inputs and outputs being in fractional Q31 numerical format. The output elements will saturate if the dot product exceeds maximum or minimum fractional values.

Remarks

Execution Time (cycles): 319 cycles + 38 / output matrix_element. The function will saturate the output value if it exceeds maximum limits per element.

Preconditions

Matrices must be aligned such that columns of A = rows of B. resMat must have the format of rows of A, columns of B. All Matrices must be set

into structure (ROWS, COLUMNS, vector_pointer).

Example

```
#define ROW1 3
#define COL1 2
#define ROW2 2
#define COL2 2

matrix32 *resMat, *srcMat1, *srcMat2;
int32_t result[ROW1*COL2];

int32_t test_MatrixA[ROW1*COL1]=
{
    0x40000000,0x20000000,    // 0.5, 0.25
    0xD999999A,0x4CCCCCCC,    // -0.3, 0.6
    0xC0000000,0x0CCCCCCD    // -0.5 0.1
};

int32_t test_MatrixB[ROW2*COL2]=
{
    0x40000000,0x20000000,    // 0.5, 0.25
    0x0CCCCCCD,0xCCCCCCD    // 0.1, -0.4
};

matrix32 mat, mat2, mat3;
resMat=&mat;
srcMat1=&mat2;
srcMat2=&mat3;

srcMat1->row=ROW1;
srcMat1->col=COL1;
srcMat1->pMatrix=test_MatrixA;

srcMat2->col=COL2;
srcMat2->row=ROW2;
srcMat2->pMatrix=test_MatrixB;

resMat->row=ROW1; // note resulting matrix MUST have ROW1 & COL2 format
resMat->col=COL2;
resMat->pMatrix=result;

DSP_MatrixMul32(resMat, srcMat1, srcMat2);

// result[] = matA[] x matB[] =
// { 0x23333333, 0x03333333 // 0.275, 0.025
//   0xF47AE147, 0xD7AE147B // -0.9, -0.315
//   0xE147AE14, 0xEAE147AE } // -0.24, -0.165
```

Parameters

Parameters	Description
resMat	pointer to different Matrix C structure (*int32_t)
srcMat1	pointer to the Matrix A structure (*int32_t)
srcMat2	pointer to the Matrix B structure (*int32_t)

DSP_MatrixScale32 Function

Scales each element of an input buffer (matrix) by a fixed number.

File

[dsp.h](#)

C

```
void DSP_MatrixScale32(int32_t * data_buffer, int N, int32_t num);
```

Returns

None.

Description

Function DSP_MatrixScale32:

```
void DSP_MatrixScale32(int32_t *data_buffer, int N, int32_t num);
```

Multiply the first N elements of an input buffer by a fixed scalar num. The resulting value is stored back into the input buffer. N number total samples of the input buffer are processed. All values are in Q31 fractional integer format. The result of calculations is saturated to the [MAX32](#) or [MIN32](#) value if exceeded.

Remarks

Execution time (cycles): 190 + 9 cycles / element, typical.

Preconditions

data_buffer must be predefined to be equal to or greater than N elements. N must be a factor of four, or will truncate to the nearest factor of four.

Example

```
int32_t numScale = 0x40000000; // 0.5
int valN = 12;
int32_t inputBufScale[12] = {0x40000000, 0x40000000, 0x20000000, 0x20000000,
    0x19999999, 0xCCCCCD, 0xF3333333, 0x80000000,
    0x7FFFFFFF, 0x00000000, 0x40000000, 0x70000000 };
// 0.5, 0.5, 0.25, 0.25, 0.25, 0.2, -0.4, -0.1, -1, 1, 0, 0.5, 0.875

DSP_MatrixScale32(inputBufScale, valN, numScale);

// inputBufScale[i] = {0x20000000, 0x20000000, 0x10000000, 0x10000000,
//     0x0CCCCC, 0xE6666666, 0xF9999999, 0xC0000000,
//     0x3FFFFFFF, 0x00000000, 0x20000000, 0x38000000}
// 0.25, 0.25, 0.125, 0.125, 0.1, -0.2, -0.05, -0.5, 0.5, 0, 0.25, 0.4375
```

Parameters

Parameters	Description
data_buffer	pointer to the Matrix to be initialized (int32_t[M*N])
N	number of elements to be initialized (int)
num	value to be initialized into the matrix (int32_t)

DSP_MatrixSub32 Function

Subtraction of two matrices $C = (A - B)$.

File

[dsp.h](#)

C

```
void DSP_MatrixSub32(matrix32 * resMat, matrix32 * srcMat1, matrix32 * srcMat2);
```

Returns

None.

Description

Function DSP_MatrixSub32:

```
void DSP_MatrixSub32(matrix32 *resMat, matrix32 *srcMat1, matrix32 *srcMat2);
```

Vector subtraction of two matrices, where both have 32-bit integer elements. The resulting output will saturate if element addition exceeds [MAX32](#) or [MIN32](#).

Remarks

Execution Time (cycles): 222 cycles + 21 / matrix_element. The function will saturate the output value if it exceeds maximum limits per element.

Preconditions

Both matrices must be equivalent in rows and columns. All Matrices must be set into structure (ROWS, COLUMNS, vector_pointer)

Example

```
#define ROW 2
```

```

#define COL 2

matrix32 *resMat, *srcMat1, *srcMat2;
int32_t result[ROW*COL];

int32_t matA[ROW*COL] = {5,2,-3,8};
int32_t matB[ROW*COL] = {2,2,2,2};

matrix32 mat, mat2, mat3;
resMat=&mat;
srcMat1=&mat2;
srcMat2=&mat3;

srcMat1->row=ROW;
srcMat1->col=COL;
srcMat1->pMatrix=matA;

srcMat2->col=COL;
srcMat2->row=ROW;
srcMat2->pMatrix=matB;

resMat->row=ROW;
resMat->col=COL;
resMat->pMatrix=result;

DSP_MatrixSub32(resMat, srcMat1, srcMat2);

// result[i] = matA[i] - matB[i] = {3,0,-5,6}

```

Parameters

Parameters	Description
resMat	pointer to different Matrix C structure (*int32_t)
srcMat1	pointer to the Matrix A structure (*int32_t)
srcMat2	pointer to the Matrix B structure (*int32_t)

DSP_MatrixTranspose32 Function

Transpose of a Matrix $C = A (T)$.

File

dsp.h

C

```
void DSP_MatrixTranspose32(matrix32 * desMat, matrix32 * srcMat);
```

Returns

None.

Description

Function DSP_MatrixTranspose32:

```
void DSP_MatrixTranspose32(matrix32 *desMat, matrix32 *srcMat);
```

Transpose of rows and columns of a matrix.

Remarks

Execution Time (cycles): 210 cycles + 10 / matrix_element.

Preconditions

Matrix definitions for ROWS and COLS must be transposed prior to the function call.

Example

```

#define ROW 3
#define COL 4

matrix32 *resMat, *srcMat1;

```

```

int32_t result[ROW*COL];

int32_t matA[ROW*COL] = { 1,  2,  3,  4,
                          5,  6,  7,  8,
                          -1, -3, -5, -7};

matrix32 mat, mat2;
resMat=&mat;
srcMat1=&mat2;

srcMat1->row=ROW;
srcMat1->col=COL;
srcMat1->pMatrix=matA;

resMat->row=COL; // note the shift in columns and rows
resMat->col=ROW;
resMat->pMatrix=result;

DSP_MatrixTranspose32(resMat, srcMat1);

// result[] = matA(T)[] = {  1,  5, -1,
//                          2,  6, -3,
//                          3,  7, -5,
//                          4,  8, -7}

```

Parameters

Parameters	Description
desMat	pointer to transposed new Matrix C (*int32_t)
srcMat	pointer to the Matrix A structure (*int32_t)

d) Transform Functions

DSP_TransformFFT16 Function

Creates an Fast Fourier Transform (FFT) from a time domain input.

File

[dsp.h](#)

C

```
void DSP_TransformFFT16(int16c * dout, int16c * din, int16c * twiddles, int16c * scratch, int log2N);
```

Returns

None.

Description

Function DSP_TransformFFT16:

```
void DSP_TransformFFT16(int16c *dout, int16c *din, int16c *twiddles, int16c *scratch, int log2N);
```

Performs an complex FFT on the input, din, and stores the complex result in dout. Performs $2^{\log_2 N}$ point calculation, and the working buffer scratch as well as the input and output must be $2^{\log_2 N}$ in length. Coefficient twiddle factors come from twiddles, and may be loaded with the use of [DSP_TransformFFT16_setup](#). All values are 16 bit (Q15) fractional.

Remarks

Scratch must be declared but need not be initialized. Din may be aided with a window function prior to calling the FFT, but is not required. Din is a complex number array, but may be loaded solely with real numbers if no phase information is available.

Preconditions

din, dout, twiddles and scratch must have N elements N is calculated as $2^{\log_2 N}$ $\log_2 N$ must be ≥ 3 FFT factors must be calculated in advance, use [DSP_TransformFFT16_setup](#)

Example

```
int log2N = 8; // log2(256) = 8
int fftSamples = 256;
```

```

int16c *fftDin;
int16c  fftDout[fftSamples];
int16c  scratch[fftSamples];
int16c  fftCoefs[fftSamples];

int16c *fftc;
fftc = &fftCoefs;

DSP_TransformFFT16_setup(fftc, log2N); // call setup function

while (1)
{
    fftDin = &fftin_8Khz_long_window16; // get 256 point complex data

    DSP_TransformFFT16(fftDout, fftDin, fftc, scratch, log2N);

    // do something with the output, fftDout
};

```

Parameters

Parameters	Description
dout	pointer to complex output array (int16c)
din	pointer to complex input array (int16c)
twiddles	pointer to an complex array of factors (int16c)
scratch	pointer to a complex scratch pad buffer (int16c)
log2N	binary exponent of number of samples (int)

DSP_TransformFFT16_setup Function

Creates FFT coefficients for use in the FFT16 function.

File

[dsp.h](#)

C

```
void DSP_TransformFFT16_setup(int16c * twiddles, int log2N);
```

Returns

None.

Description

Function DSP_TransformFFT16_setup:

```
void DSP_TransformFFT16_setup(int16c *twiddles, int log2N);
```

Calculates the N twiddle factors required to operate the FFT16 function. These factors are done in serial fashion, and require considerable processing power. Ideally this function would be run only once prior to an ongoing FFT, and the results held in a buffer.

Remarks

This function is of considerable length and executed in C. It is recommended it only be called once for any given FFT length in time sensitive applications.

Preconditions

twiddles must be N in length N is calculated ($2^{\log 2N}$)

Example

see DSP_TransformFFT16 [for](#) example.

Parameters

Parameters	Description
twiddles	pointer to a complex array of factors (int16c)
log2N	binary exponent of number of data points (int)

DSP_TransformFFT32 Function

Creates an Fast Fourier Transform (FFT) from a time domain input.

File

[dsp.h](#)

C

```
void DSP_TransformFFT32(int32c * dout, int32c * din, int32c * twiddles, int32c * scratch, int log2N);
```

Returns

None.

Description

Function DSP_TransformFFT32:

```
void DSP_TransformFFT32(int32c *dout, int32c *din, int32c *twiddles, int32c *scratch, int log2N);
```

Performs an complex FFT on the input, din, and stores the complex result in dout. Performs $2^{\log_2 N}$ point calculation, and the working buffer scratch as well as the input and output must be $2^{\log_2 N}$ in length. Coefficient twiddle factors come from twiddles, and may be loaded with the use of [DSP_TransformFFT16_setup](#). All values are 16 bit (Q31) fractional.

Remarks

Scratch must be declared but need not be initialized. Din may be aided with a window function prior to calling the FFT, but is not required. Din is a complex number array, but may be loaded solely with real numbers if no phase information is available.

Preconditions

din, dout, twiddles and scratch must have N elements N is calculated as $2^{\log_2 N}$ $\log_2 N$ must be ≥ 3 FFT factors must be calculated in advance, use [DSP_TransformFFT32_setup](#)

Example

```
int log2N = 8; // log2(256) = 8
int fftSamples = 256;

int32c *fftDin;
int32c fftDout[fftSamples];
int32c scratch[fftSamples];
int32c fftCoefs[fftSamples];

int32c *fftc;
fftc = &fftCoefs;

DSP_TransformFFT32_setup(fftc, log2N); // call setup function

while (1)
{
    fftDin = &fftin_5Khz_long_window32; // get 256 point complex data

    DSP_TransformFFT32(fftDout, fftDin, fftc, scratch, log2N);

    // do something with the output, fftDout
};
```

Parameters

Parameters	Description
dout	pointer to complex output array (int32c)
din	pointer to complex input array (int32c)
twiddles	pointer to an complex array of FFT factors (int32c)
scratch	pointer to a complex scratch pad buffer (int32c)
log2N	binary exponent of number of samples (int)

DSP_TransformFFT32_setup Function

Creates FFT coefficients for use in the FFT32 function.

File

[dsp.h](#)

C

```
void DSP_TransformFFT32_setup(int32c * twiddles, int log2N);
```

Returns

None.

Description

Function DSP_TransformFFT32_setup:

```
void DSP_TransformFFT32_setup(int32c *twiddles, int log2N);
```

Calculates the N FFT twiddle factors required to operate the FFT32 function. These factors are done in serial fashion, and require considerable processing power. Ideally this function would be run only once prior to an ongoing FFT, and the results held in a buffer.

Remarks

This function is of considerable length and executed in C. It is recommended it only be called once for any given FFT length in time sensitive applications.

Preconditions

twiddles must be N in length N is calculated ($2^{\log 2N}$)

Example

see DSP_TransformFFT32 [for](#) example.

Parameters

Parameters	Description
twiddles	pointer to a complex array of coefficients (int32c)
log2N	binary exponent of number of data points (int)

DSP_TransformIFFT16 Function

Creates an Inverse Fast Fourier Transform (FFT) from a frequency domain input.

File

[dsp.h](#)

C

```
void DSP_TransformIFFT16(int16c * dout, int16c * din, int16c * twiddles, int16c * scratch, int log2N);
```

Returns

None.

Description

Function DSP_TransformIFFT16:

```
void DSP_TransformIFFT16(int16c *dout, int16c *din, int16c *twiddles, int16c *scratch, int log2N);
```

Performs a complex Inverse FFT on the input, din, and stores the complex result in dout. Performs $2^{\log 2N}$ point calculation, and the working buffer scratch as well as the input and output must be $2^{\log 2N}$ in length. Coefficient twiddle factors come from twiddles, and may be loaded with the use of [DSP_TransformFFT16_setup](#). All values are 16 bit (Q15) fractional.

Remarks

Scratch must be declared but need not be initialized. Din may be aided with a window function prior to calling the FFT, but is not required. A very similar function to the FFT is executed for the inverse FFT. This requires twiddle factors set in advance with the same method as used in the FFT. Complex conjugate and scaling are handled within the algorithm. The output is scaled using binary shifting based on log2N. Since the algorithm reduces the output by a scale factor of log2N, the resolution is reduced proportionally to the number of data points.

Preconditions

din, dout, twiddles and scratch must have N elements N is calculated as $2^{(\log 2N)}$ log2N must be ≥ 3 FFT factors must be calculated in advance, use [DSP_TransformFFT16_setup](#)

Example

```

int ilog2N = 10; // log2(64) = 6; log2(256) = 8; log2(1024) = 10;
int ifftSamples = pow(2,ilog2N);

int16c *ifftDin;
int16c ifftDout[ifftSamples];
int16c iscratch[ifftSamples];
int16c ifftCoefs[ifftSamples];
int16c ifftTimeOut[ifftSamples];

// set up twiddle factors, these are used for both FFT and iFFT

int16c *ifftc;
ifftc = &ifftCoefs;
DSP_TransformFFT16_setup( ifftc, ilog2N); // call to coef setup

// in this example, we take an FFT of an original time domain (sine wave)
// the output of the FFT is used as the input of the iFFT for comparison

ifftDin = &fftin_800hz_verylong16;
DSP_TransformFFT16(ifftDout, ifftDin, ifftc, iscratch, ilog2N);

// ifftDout = frequency domain output, complex number array

DSP_TransformIFFT16(ifftTimeOut, ifftDout, ifftc, iscratch, ilog2N);

// do something with the output, fftTimeOut, time domain

```

Parameters

Parameters	Description
dout	pointer to complex output array (int16c)
din	pointer to complex input array (int16c)
twiddles	pointer to an complex array of factors (int16c)
scratch	pointer to a complex scratch pad buffer (int16c)
log2N	binary exponent of number of samples (int)

DSP_TransformWindow_Bart16 Function

Perform a Bartlett window on a vector.

File

[dsp.h](#)

C

```
void DSP_TransformWindow_Bart16(int16_t * OutVector, int16_t * InVector, int N);
```

Returns

None.

Description

Function DSP_TransformWindow_Bart16:

```
void DSP_TransformWindow_Bart16(int16_t *OutVector, int16_t *InVector, int N);
```

Compute a Bartlett (Triangle) Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q15 fractional format. The Bartlett Window follows the equation:

Window(n) = 1 - (abs(2*n - N)/N) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

```
int16_t OutVector16[8]={0};
int WindowN = 8;

for (i=0;i<WindowN;i++)
{
    InVector16[i]= 0x4000; // constant 0.5 for functional testing
}

DSP_TransformWindow_Bart16(OutVector16, InVector16, WindowN);
// OutWindow = 0x0000, 0x1000, 0x2000, 0x3000, 0x4000, 0x3000, 0x2000, 0x1000
```

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

DSP_TransformWindow_Bart32 Function

Perform a Bartlett window on a vector.

File

[dsp.h](#)

C

```
void DSP_TransformWindow_Bart32(int32_t * OutVector, int32_t * InVector, int N);
```

Returns

None.

Description

Function DSP_TransformWindow_Bart32:

```
void DSP_TransformWindow_Bart32(int32_t *OutVector, int32_t *InVector, int N);
```

Compute a Bartlett (Triangle) Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q31 fractional format. The Bartlett Window follows the equation:

Window(n) = 1 - (abs(2*n - N)/N) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

```
int32_t OutVector32[8]={0};
int WindowN = 8;

for (i=0;i<WindowN;i++)
{
    InVector32[i]= 0x40000000; // constant 0.5 for functional testing
}

DSP_TransformWindow_Bart32(OutVector32, InVector32, WindowN);
// OutWindow = 0x0, 0x10000000, 0x20000000, 0x30000000, 0x40000000,
//              0x30000000, 0x20000000, 0x10000000
```


Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

DSP_TransformWindow_Black16 Function

Perform a Blackman window on a vector.

File

[dsp.h](#)

C

```
void DSP_TransformWindow_Black16(int16_t * OutVector, int16_t * InVector, int N);
```

Returns

None.

Description

Function DSP_TransformWindow_Black16:

```
void DSP_TransformWindow_Black16(int16_t *OutVector, int16_t *InVector, int N);
```

Compute a Blackman Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q15 fractional format. The Blackman Window follows the equation:

Window(n) = 0.42659 - 0.49656 * COS(2*Pi*n/(N-1)) + 0.076849 * COS(4*Pi*n/(N-1)) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

```
int16_t OutVector16[8]={0};
int WindowN = 8;

for (i=0;i<WindowN;i++)
{
    InVector16[i]= 0x4000; // constant 0.5 for functional testing
}

DSP_TransformWindow_Black16(OutVector16, InVector16, WindowN);
// OutWindow = 0x0071, 0x0665, 0x1DF1, 0x3B00, 0x3B00, 0x1DF1, 0x0665, 0x0071
```

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

DSP_TransformWindow_Black32 Function

Perform a Blackman window on a vector.

File

[dsp.h](#)

C

```
void DSP_TransformWindow_Black32(int32_t * OutVector, int32_t * InVector, int N);
```

Returns

None.

Description

Function DSP_TransformWindow_Black32:

```
void DSP_TransformWindow_Black32(int32_t *OutVector, int32_t *InVector, int N);
```

Compute a Blackman Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q31 fractional format. The Blackman Window follows the equation:

$Window(n) = 0.42659 - 0.49656 * \cos(2 * \pi * n / (N - 1)) + 0.076849 * \cos(4 * \pi * n / (N - 1))$ where n is the window sample number N is the total number of samples The functional output computes $WinVector(n) = Window(n) * InVector(n)$

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

```
int32_t OutVector32[8]={0};
int WindowN = 8;

for (i=0;i<WindowN;i++)
{
    InVector32[i]= 0x40000000; // constant 0.5 for functional testing
}

DSP_TransformWindow_Black32(OutVector32, InVector32, WindowN);
// OutWindow = 0x0070B490, 0x06649680, 0x1DF13240, 0x3B003D80, 0x3B003D80,
//              0x1DF13240, 0x06649680, 0x0070B490
```

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

DSP_TransformWindow_Cosine16 Function

Perform a Cosine (Sine) window on a vector.

File

[dsp.h](#)

C

```
void DSP_TransformWindow_Cosine16(int16_t * OutVector, int16_t * InVector, int N);
```

Returns

None.

Description

Function DSP_TransformWindow_Cosine16:

```
void DSP_TransformWindow_Cosine16(int16_t *OutVector, int16_t *InVector, int N);
```

Compute a Cosine Window on the first N samples of the input vector, InVector. The output is stored in OutWindow. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q15 fractional format. The Cosine Window follows the equation:

$Window(n) = \sin(\pi * n / (N - 1))$ where n is the window sample number N is the total number of samples The functional output computes $WinVector(n) = Window(n) * InVector(n)$

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

```
int16_t OutVector16[8]={0};
int WindowN = 8;

for (i=0;i<WindowN;i++)
{
    InVector16[i]= 0x4000; // constant 0.5 for functional testing
}

DSP_TransformWindow_Cosine16(OutVector16, InVector16, WindowN);
// OutWindow = 0x0000, 0x1BC5, 0x320A, 0x3E65, 0x3E65, 0x320A, 0x1BC5, 0x0071
```

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

DSP_TransformWindow_Cosine32 Function

Perform a Cosine (Sine) window on a vector.

File

[dsp.h](#)

C

```
void DSP_TransformWindow_Cosine32(int32_t * OutVector, int32_t * InVector, int N);
```

Returns

None.

Description

Function DSP_TransformWindow_Cosine32:

```
void DSP_TransformWindow_Cosine32(int32_t *OutVector, int32_t *InVector, int N);
```

Compute a Cosine Window on the first N samples of the input vector, InVector. The output is stored in OutWindow. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q31 fractional format. The Cosine Window follows the equation:

Window(n) = SIN(Pi*n/(N-1)) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

```
int32_t OutVector32[8]={0};
int WindowN = 8;

for (i=0;i<WindowN;i++)
{
    InVector32[i]= 0x40000000; // constant 0.5 for functional testing
}

DSP_TransformWindow_Cosine32(OutVector32, InVector32, WindowN);
// OutWindow = 0x00000000, 0x1BC4C060, 0x32098700, 0x3E653800, 0x3E653800,
// 0x32098700, 0x1BC4C060, 0x00000000
```

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

DSP_TransformWindow_Hamm16 Function

Perform a Hamming window on a vector.

File

[dsp.h](#)

C

```
void DSP_TransformWindow_Hamm16(int16_t * OutVector, int16_t * InVector, int N);
```

Returns

None.

Description

Function DSP_TransformWindow_Hamm16:

```
void DSP_TransformWindow_Hamm16(int16_t *OutVector, int16_t *InVector, int N);
```

Compute a Hamming Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q15 fractional format. The Hamming Window follows the equation:

Window(n) = 0.54 - 0.46 * COS(2*Pi*n/N) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

```
int16_t OutVector16[8]={0};
int WindowN = 8;

for (i=0;i<WindowN;i++)
{
    InVector16[i]= 0x4000; // constant 0.5 for functional testing
}

DSP_TransformWindow_Hamm16(OutVector16, InVector16, WindowN);
// OutWindow = 0x051F, 0x0DBE, 0x228F, 0x3761, 0x4000, 0x3761, 0x228F, 0x0DBE
```

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

DSP_TransformWindow_Hamm32 Function

Perform a Hamming window on a vector.

File

[dsp.h](#)

C

```
void DSP_TransformWindow_Hamm32(int32_t * OutVector, int32_t * InVector, int N);
```

Returns

None.

Description

Function DSP_TransformWindow_Hamm32:

```
void DSP_TransformWindow_Hamm32(int32_t *OutVector, int32_t *InVector, int N);
```

Compute a Hamming Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q31 fractional format. The Hamming Window follows the equation:

$Window(n) = 0.54 - 0.46 * \cos(2 * \pi * n / N)$ where n is the window sample number N is the total number of samples The functional output computes $WinVector(n) = Window(n) * InVector(n)$

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

```
int32_t OutVector32[8]={0};
int WindowN = 8;

for (i=0;i<WindowN;i++)
{
    InVector32[i]= 0x40000000; // constant 0.5 for functional testing
}

DSP_TransformWindow_Hamm32(OutVector32, InVector32, WindowN);
// OutWindow = 0x051EB860, 0x0DBE26C0, 0x228F5C40, 0x37609200, 0x40000000,
//              0x37609200, 0x228F5C40, 0x0DBE26C0
```

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

DSP_TransformWindow_Hann16 Function

Perform a Hanning window on a vector.

File

[dsp.h](#)

C

```
void DSP_TransformWindow_Hann16(int16_t * OutVector, int16_t * InVector, int N);
```

Returns

None.

Description

Function DSP_TransformWindow_Hann16:

```
void DSP_TransformWindow_Hann16(int16_t *OutVector, int16_t *InVector, int N);
```

Compute a Hanning Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q15 fractional format. The Hanning Window follows the equation:

$Window(n) = 0.5 - 0.5 * \cos(2 * \pi * n / N)$ where n is the window sample number N is the total number of samples The functional output computes $WinVector(n) = Window(n) * InVector(n)$

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

```
int16_t OutVector16[8]={0};
int WindowN = 8;

for (i=0;i<WindowN;i++)
{
    InVector16[i]= 0x4000; // constant 0.5 for functional testing
}

DSP_TransformWindow_Hann16(OutVector16, InVector16, WindowN);
// OutWindow = 0x0000, 0x095F, 0x2000, 0x36A1, 0x4000, 0x36A1, 0x2000, 0x095F
```

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

DSP_TransformWindow_Hann32 Function

Perform a Hanning window on a vector.

File

[dsp.h](#)

C

```
void DSP_TransformWindow_Hann32(int32_t * OutVector, int32_t * InVector, int N);
```

Returns

None.

Description

Function DSP_TransformWindow_Hann32:

```
void DSP_TransformWindow_Hann32(int32_t *OutVector, int32_t *InVector, int N);
```

Compute a Hanning Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q31 fractional format. The Hanning Window follows the equation:

Window(n) = 0.5 - 0.5 * COS(2*Pi*n/N) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

```
int32_t OutVector32[8]={0};
int WindowN = 8;

for (i=0;i<WindowN;i++)
{
    InVector32[i]= 0x40000000; // constant 0.5 for functional testing
}

DSP_TransformWindow_Hann32(OutVector32, InVector32, WindowN);
// OutWindow = 0x00000000, 0x095F61C0, 0x20000000, 0x36A09E80, 0x40000000,
// 0x36A09E80, 0x20000000, 0x095F61C0
```

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

DSP_TransformWindow_Kaiser16 Function

Perform a Kaiser window on a vector.

File

[dsp.h](#)

C

```
void DSP_TransformWindow_Kaiser16(int16_t * OutVector, int16_t * InVector, int N);
```

Returns

None.

Description

Function DSP_TransformWindow_Kaiser16:

```
void DSP_TransformWindow_Kaiser16(int16_t *OutVector, int16_t *InVector, int N);
```

Compute a Kaiser Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q15 fractional format. The Kaiser Window follows the equation:

Window(n) = 0.402 - 0.498 * COS(2*Pi*n/N) + 0.098 * cos(4*Pi*n/N) + 0.001 * cos(6*Pi*n/N) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

```
int16_t OutVector16[8]={0};
int WindowN = 8;

for (i=0;i<WindowN;i++)
{
    InVector16[i]= 0x4000; // constant 0.5 for functional testing
}

DSP_TransformWindow_Kaiser16(OutVector16, InVector16, WindowN);
// OutWindow = 0x0031, 0x0325, 0x1375, 0x304F, 0x3FCF, 0x304F, 0x1375, 0x0325
```

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

DSP_TransformWindow_Kaiser32 Function

Perform a Kaiser window on a vector.

File

[dsp.h](#)

C

```
void DSP_TransformWindow_Kaiser32(int32_t * OutVector, int32_t * InVector, int N);
```

Returns

None.

Description

Function DSP_TransformWindow_Kaiser32:

```
void DSP_TransformWindow_Kaiser32(int32_t *OutVector, int32_t *InVector, int N);
```

Compute a Kaiser Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q31 fractional format. The Kaiser Window follows the equation:

Window(n) = 0.402 - 0.498 * COS(2*Pi*n/N) + 0.098 * cos(4*Pi*n/N) + 0.001 * cos(6*Pi*n/N) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

```
int32_t OutVector32[8]={0};
int WindowN = 8;

for (i=0;i<WindowN;i++)
{
    InVector32[i]= 0x40000000; // constant 0.5 for functional testing
}

DSP_TransformWindow_Kaiser32(OutVector32, InVector32, WindowN);
// OutWindow = 0x003126F6, 0x032555C8, 0x1374BCA0, 0x304F66C0, 0x3FCED900,
//             0x304F66C0, 0x1374BCA0, 0x032555C8
```

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

DSP_TransformWinInit_Bart16 Function

Create a Bartlett window.

File

[dsp.h](#)

C

```
void DSP_TransformWinInit_Bart16(int16_t * OutWindow, int N);
```

Returns

None.

Description

Function DSP_TransformWinInit_Bart16:

```
void DSP_TransformWinInit_Bart16(int16_t *OutWindow, int N);
```

Create a N-element Bartlett (Triangle) Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q15 fractional format. The Bartlett Window follows the equation:

Window(n) = 1 - (abs(2*n - N)/N) where n is the window sample number N is the total number of samples

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

DSP_TransformWinInit_Bart32 Function

Create a Bartlett window.

File

[dsp.h](#)

C

```
void DSP_TransformWinInit_Bart32(int32_t * OutWindow, int N);
```

Returns

None.

Description

Function DSP_TransformWinInit_Bart32:

```
void DSP_TransformWinInit_Bart32(int32_t *OutWindow, int N);
```

Create a N-element Bartlett (Triangle) Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values Q31 fractional format. The Bartlett Window follows the equation:

Window(n) = 1 - (abs(2*n - N)/N) where n is the window sample number N is the total number of samples

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

DSP_TransformWinInit_Black16 Function

Create a Blackman window.

File

[dsp.h](#)

C

```
void DSP_TransformWinInit_Black16(int16_t * OutWindow, int N);
```

Returns

None.

Description

Function DSP_TransformWinInit_Black16:

```
void DSP_TransformWinInit_Black16(int16_t *OutWindow, int N);
```

Create a N-element Blackman Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are Q16 fractional format. The Blackman Window follows the equation:

Window(n) = 0.42659 - 0.49656 * COS(2*Pi*n/(N-1)) + 0.076849 * COS(4*Pi*n/(N-1)) where n is the window sample number N is the total number of samples

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

DSP_TransformWinInit_Black32 Function

Create a Blackman window.

File

[dsp.h](#)

C

```
void DSP_TransformWinInit_Black32(int32_t * OutWindow, int N);
```

Returns

None.

Description

Function DSP_TransformWinInit_Black32:

```
void DSP_TransformWinInit_Black32(int32_t *OutWindow, int N);
```

Create a N-element Blackman Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are Q31 fractional format. The Blackman Window follows the equation:

Window(n) = 0.42659 - 0.49656 * COS(2*Pi*n/(N-1)) + 0.076849 * COS(4*Pi*n/(N-1)) where n is the window sample number N is the total number of samples

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

DSP_TransformWinInit_Cosine16 Function

Create a Cosine (Sine) window.

File

[dsp.h](#)

C

```
void DSP_TransformWinInit_Cosine16(int16_t * OutWindow, int N);
```

Returns

None.

Description

Function DSP_TransformWinInit_Cosine16:

```
void DSP_TransformWinInit_Cosine16(int16_t *OutWindow, int N);
```

Create a N-element Cosine Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q15 fractional format. The Cosine Window follows the equation:

Window(n) = SIN(Pi*n/(N-1)) where n is the window sample number N is the total number of samples

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

DSP_TransformWinInit_Cosine32 Function

Create a Cosine (Sine) window.

File

[dsp.h](#)

C

```
void DSP_TransformWinInit_Cosine32(int32_t * OutWindow, int N);
```

Returns

None.

Description

Function DSP_TransformWinInit_Cosine32:

```
void DSP_TransformWinInit_Cosine32(int32_t *OutWindow, int N);
```

Create a N-element Cosine Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q31 fractional format. The Cosine Window follows the equation:

Window(n) = SIN(Pi*n/(N-1)) where n is the window sample number N is the total number of samples

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

DSP_TransformWinInit_Hamm16 Function

Create a Hamming window.

File

[dsp.h](#)

C

```
void DSP_TransformWinInit_Hamm16(int16_t * OutWindow, int N);
```

Returns

None.

Description

Function DSP_TransformWinInit_Hamm16:

```
void DSP_TransformWinInit_Hamm16(int16_t *OutWindow, int N);
```

Create a N-element Hamming Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q15 fractional format. The Hamming Window follows the equation:

Window(n) = 0.54 - 0.46 * COS(2*Pi*n/N) where n is the window sample number N is the total number of samples

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

DSP_TransformWinInit_Hamm32 Function

Create a Hamming window.

File

[dsp.h](#)

C

```
void DSP_TransformWinInit_Hamm32(int32_t * OutWindow, int N);
```

Returns

None.

Description

Function DSP_TransformWinInit_Hamm32:

```
void DSP_TransformWinInit_Hamm32(int32_t *OutWindow, int N);
```

Create a N-element Hamming Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q31 fractional format. The Hamming Window follows the equation:

Window(n) = 0.54 - 0.46 * COS(2*Pi*n/N) where n is the window sample number N is the total number of samples

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

DSP_TransformWinInit_Hann16 Function

Create a Hanning window.

File

[dsp.h](#)

C

```
void DSP_TransformWinInit_Hann16(int16_t * OutWindow, int N);
```

Returns

None.

Description

Function DSP_TransformWinInit_Hann16:

```
void DSP_TransformWinInit_Hann16(int16_t *OutWindow, int N);
```

Create a N-element Hanning Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q15 fractional format. The Hanning Window follows the equation:

Window(n) = 0.5 - 0.5 * COS(2*Pi*n/N) where n is the window sample number N is the total number of samples

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

DSP_TransformWinInit_Hann32 Function

Create a Hanning window.

File

[dsp.h](#)

C

```
void DSP_TransformWinInit_Hann32(int32_t * OutWindow, int N);
```

Returns

None.

Description

Function DSP_TransformWinInit_Hann32:

```
void DSP_TransformWinInit_Hann32(int32_t *OutWindow, int N);
```

Create a N-element Hanning Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q31 fractional format. The Hanning Window follows the equation:

Window(n) = 0.5 - 0.5 * COS(2*Pi*n/N) where n is the window sample number N is the total number of samples

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

DSP_TransformWinInit_Kaiser16 Function

Create a Kaiser window.

File

[dsp.h](#)

C

```
void DSP_TransformWinInit_Kaiser16(int16_t * OutWindow, int N);
```

Returns

None.

Description

Function DSP_TransformWinInit_Kaiser16:

```
void DSP_TransformWinInit_Kaiser16(int16_t *OutWindow, int N);
```

Create a N-element Kaiser Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q15 fractional format. The Kaiser Window follows the equation:

Window(n) = 0.402 - 0.498 * COS(2*Pi*n/N) + 0.098 * cos(4*Pi*n/N) + 0.001 * cos(6*Pi*n/N) where n is the window sample number N is the total number of samples

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the

window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

DSP_TransformWinInit_Kaiser32 Function

Create a Kaiser window.

File

[dsp.h](#)

C

```
void DSP_TransformWinInit_Kaiser32(int32_t * OutWindow, int N);
```

Returns

None.

Description

Function DSP_TransformWinInit_Kaiser32:

```
void DSP_TransformWinInit_Kaiser32(int32_t *OutWindow, int N);
```

Create a N-element Kaiser Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q31 fractional format. The Kaiser Window follows the equation:

Window(n) = 0.402 - 0.498 * COS(2*Pi*n/N) + 0.098 * cos(4*Pi*n/N) + 0.001 * cos(6*Pi*n/N) where n is the window sample number N is the total number of samples

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Example

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

e) Vector Math Functions

DSP_VectorAbs16 Function

Calculate the absolute value of a vector.

File

[dsp.h](#)

C

```
void DSP_VectorAbs16(int16_t * outdata, int16_t * indata, int N);
```

Returns

None.

Description

Function DSP_VectorAbs16:

```
void DSP_VectorAbs16(int16_t *outdata, int16_t *indata, int N);
```

Computes the absolute value of each element of indata and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q15 fractional format. outdata[i] filled with the absolute value of elements of indata

Remarks

This must be assembled with .set microMIPS.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Example

```
int16_t *pOutdata;
int16_t outVal[8];
int16_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100,200,127,-127,-2};
int Num = 8;

pOutdata = &outVal;

DSP_VectorAbs16(pOutdata, inBufTest, Num);

// outVal[i] = {5,2,3,4,1,0,2,8}
```

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int16_t)
indata	pointer to input array of 16-bit elements (int16_t)
N	number of samples (int)

DSP_VectorAbs32 Function

Calculate the absolute value of a vector.

File

[dsp.h](#)

C

```
void DSP_VectorAbs32(int32_t * outdata, int32_t * indata, int N);
```

Returns

None.

Description

Function DSP_VectorAbs32:

```
void DSP_VectorAbs32(int32_t *outdata, int32_t *indata, int N);
```

Computes the absolute value of each element of indata and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q31 fractional format. outdata[i] filled with N elements of abs(indata[i])

Remarks

This must be assembled with .set microMIPS.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Example

```
int16_t *pOutdata;
int32_t outVal[8];
int32_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100,200,127,-127,-2};
int Num = 8;

pOutdata = &outVal;

DSP_VectorAbs32(pOutdata, inBufTest, Num);

// outVal[i] = {5,2,3,4,1,0,2,8}
```

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int32_t)
indata	pointer to input array of 16-bit elements (int32_t)
N	number of samples (int)

DSP_VectorAdd16 Function

Calculate the sum of two vectors.

File

[dsp.h](#)

C

```
void DSP_VectorAdd16(int16_t * outdata, int16_t * indata1, int16_t * indata2, int N);
```

Returns

None.

Description

Function DSP_VectorAdd16:

```
void DSP_VectorAdd16(int16_t *outdata, int16_t *indata1, int16_t *indata2, int N);
```

Computes the sum value of each element of indata1 + indata2 and stores it to outdata. The number of samples to process is given by the parameter N. Data is in the Q15 fractional format. outdata[i] filled with N elements of indata1[i] + indata2[i]

Remarks

This must be assembled with .set microMIPS.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Example

```
int16_t *pOutdata;
int16_t outVal[8];
int16_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100,200,127,-127,-2};
int16_t inBuf2[16]= {1,2,3,4,5,6,7,8,9,10,-1,-100,-127,127,-7,0};
int Num = 8;

pOutdata = &outVal;

DSP_VectorAdd16(pOutdata, inBufTest, inBuf2, Num);

// outVal[i] = inBufTest[i] + inBuf2[i] = {-4,4,0,8,4,6,5,0}
```

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int16_t)
indata1	pointer to input array of 16-bit elements (int16_t)
indata2	pointer to input array of 16-bit elements (int16_t)

N	number of samples (int)
---	-------------------------

DSP_VectorAdd32 Function

Calculate the sum of two vectors.

File

[dsp.h](#)

C

```
void DSP_VectorAdd32(int32_t * outdata, int32_t * indata1, int32_t * indata2, int N);
```

Returns

None.

Description

Function DSP_VectorAdd32:

```
void DSP_VectorAdd32(int32_t *outdata, int32_t *indata1, int32_t *indata2, int N);
```

Computes the sum value of each element of indata1 + indata2 and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q31 fractional format. outdata[i] filled with N elements of indata1[i] + indata2[i]

Remarks

This must be assembled with .set microMIPS.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Example

```
int16_t *pOutdata;
int32_t outVal[8];
int32_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100, 200, 127,-127,-2};
int32_t inBuf2[16]=    { 1,2, 3,4, 5,6, 7, 8, 9, 10,-1,-100,-127,127,-7, 0};
int Num = 8;

pOutdata = &outVal;

DSP_VectorAdd32(pOutdata, inBufTest, inBuf2, Num);

// outVal[i] = inBufTest[i] + inBuf2[i] = {-4,4,0,8,4,6,5,0}
```

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int32_t)
indata1	pointer to input array of 16-bit elements (int32_t)
indata2	pointer to input array of 16-bit elements (int32_t)
N	number of samples

DSP_VectorAddc16 Function

Calculate the sum of a vector and a constant.

File

[dsp.h](#)

C

```
void DSP_VectorAddc16(int16_t * outdata, int16_t * indata, int16_t c, int N);
```

Returns

None.

Description

Function DSP_VectorAddc16:

```
void DSP_VectorAddc16(int16_t *outdata, int16_t *indata, int16_t c, int N);
```

Computes the sum value of each element of (indata + c) and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q15 fractional format. outdata[i] filled with N elements of indata[i] + c

Remarks

This must be assembled with .set microMIPS.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Example

```
int16_t *pOutdata;
int16_t outVal[8];
int16_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100, 200, 127,-127,-2};
int16_t constValue = 3;
int Num = 8;

pOutdata = &outVal;

DSP_VectorAddc16(pOutdata, inBufTest, constValue, Num);

// outVal[i] = inBufTest[i] + constValue = {-2,5,0,7,2,3,1,-5}
```

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int16_t)
indata	pointer to input array of 16-bit elements (int16_t)
c	constant value added to all indata1 elements (int16_t)
N	number of samples (int)

DSP_VectorAddc32 Function

Calculate the sum of a vector and a constant.

File

[dsp.h](#)

C

```
void DSP_VectorAddc32(int32_t * outdata, int32_t * indata, int32_t c, int N);
```

Returns

None.

Description

Function DSP_VectorAddc32:

```
void DSP_VectorAddc32(int32_t *outdata, int32_t *indata, int32_t c, int N);
```

Computes the sum value of each element of (indata + c) and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q31 fractional format. outdata[i] filled with N elements of indata1[i] + c

Remarks

This must be assembled with .set microMIPS.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Example

```
int16_t *pOutdata;
int32_t outVal[8];
int32_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100, 200, 127,-127,-2};
```

```
int32_t constValue = 3;
int Num = 8;

pOutdata = &outVal;

DSP_VectorAddc32(pOutdata, inBufTest, constValue, Num);

// outVal[i] = inBufTest[i] + constValue = {-2,5,0,7,2,3,1,-5}
```

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int32_t)
indata	pointer to input array of 16-bit elements (int32_t)
c	constant value added to all indata1 elements (int32_t)
N	number of samples (int)

DSP_VectorAutocorr16 Function

Computes the Autocorrelation of a Vector.

File

[dsp.h](#)

C

```
void DSP_VectorAutocorr16(int16_t * outCorr, int16_t * inVector, int N, int K);
```

Returns

None.

Description

Function DSP_VectorAutocorr16:

```
void DSP_VectorAutocorr16(int16_t *outCorr, int16_t *inVector, int N, int K);
```

Calculates the autocorrelation, with a lag of 1 to K, on the first N elements of inVector and returns the 16-bit scalar result in outCorr. The autocorrelation is calculated from other statistical equations including mean and variance. While in some cases these equations exist inside the DSP library, the functions are executed in a serial fashion within this code to provide enhanced performance. The unbiased function has the form - mean (M) = $\sum[0..N-1](x(n) / N)$ variance (V) = $\sum[0..N-1]((x(n) - M)^2) / (N-1)$ autocovariance (ACV)[k] = $\sum[0..(N-k)]((x(n) - M) * (x(n+k) - M) / (N-k))$ autocorrelation (AC)[k] = CV[k] / V where N is the number of vector elements, n is the index of those elements x(n) is a single element in the input vector M is the mean of the N elements of the vector k is the lag or series index

The output of the function will return K elements, and the outCorr array should be sized to accept those 16-bit results. The outputs correspond to k=1, k=2, ..., k=K delay states. The function returns a 16-bit value in rounded, saturated Q15 format.

Input values of the vector and output scalar value is Q15 fractional format. This format has data that ranges from -1 to 1, and has internal saturation limits of those same values. Some care has been taken to reduce the impact of saturation by adding processing steps to effectively complete the processing in blocks. However, in some extreme cases of data variance it is still possible to reach the saturation limits.

Remarks

This function is optimized with microMIPS and M14KCe ASE DSP instructions. This function is dependent on the LibQ library, and the [_LIBQ_Q16Div](#) specifically.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. outCorr must be an array with at least K values. N must be greater than or equal to four and a multiple of four.

Example

```
int autocvN = 16; // N value, number of samples in array
int autocvLag = 4; // Lag value, output shifts to observe
int16_t outAC[16]={0};
int16_t CVIn16[16] = {0x1999, 0xD99A, 0x1000, 0x6000, 0x1999, 0x1999, 0x2666,
    0x3333, 0x1000, 0x6000, 0x1999, 0x1999, 0x2666, 0x3333, 0x1999, 0x0CCC};
// = { .2, -.3, .125, .75, .2, .2, .3, .4, .125, .75, .2, .2, .3, .4, .2, .1};

DSP_VectorAutocorr16(outAC, CVIn16, autocvN, autocvLag);

// outAC = {0xF406, D46C, 0x098F, 0x191A}
```

```
// = -0.093567, -0.34045, 0.07468, 0.19611
```

Parameters

Parameters	Description
outCorr	pointer to output array (int16_t)
inVector	pointer to source array of elements (int16_t)
N	number of samples (int)
K	lag value, number of output elements (int)

DSP_VectorBexp16 Function

Computes the maximum binary exponent of a vector.

File

[dsp.h](#)

C

```
int DSP_VectorBexp16(int16_t * DataIn, int N);
```

Returns

Binary exponent [log2 multiplier] (int)

Description

Function DSP_VectorBexp16:

```
int DSP_VectorBexp16(int16_t *DataIn, int N);
```

Calculates the maximum binary exponent on the first N elements of the input vector DataIn, and stores the integer result. The returned value represents the potential binary scaling of the vector, and may be used with other functions that auto scale their output without saturation. Inputs are given in Q15 fractional data format.

Remarks

None.

Preconditions

N must be a multiple of 2 and greater or equal to 2.

Example

```
int valN = 4;
int16_t dummy16[valN]={0x3004, 0x00CC, 0xFC04, 0xFFF0};
//                               0.375, 0.0062, -0.0311, -0.00049
int answer;

answer = DSP_VectorBexp16(dummy16, valN);

// answer = 1, maximum binary gain is 2.
```

Parameters

Parameters	Description
DataIn	pointer to input array of 16-bit elements (int16_t)
N	number of samples (int)

DSP_VectorBexp32 Function

Computes the maximum binary exponent of a vector.

File

[dsp.h](#)

C

```
int DSP_VectorBexp32(int32_t * DataIn, int N);
```

Returns

Binary exponent [log2 multiplier] (int)

Description

Function DSP_VectorBexp32:

```
int DSP_VectorBexp32(int32_t *DataIn, int N);
```

Calculates the maximum binary exponent on the first N elements of the input vector DataIn, and stores the integer result. The returned value represents the potential binary scaling of the vector, and may be used with other functions that auto scale their output without saturation. Inputs are given in Q31 fractional data format.

Remarks

None.

Preconditions

None.

Example

```
int valN=4;
int32_t datInput32[4]={0xFF000000, 0x07000000,0x000CCCCC, 0x08000000};
//                               -0.007183, 0.054688, 0.0003906, 0.0625
int answer32;

answer32 = DSP_VectorBexp32(datInput32, valN);

// answer = 3, maximum binary gain is 8.
```

Parameters

Parameters	Description
DataIn	pointer to input array of 16-bit elements (int32_t)
N	number of samples (int)

DSP_VectorChkEqu32 Function

Compares two input vectors, returns an integer '1' if equal, and '0' if not equal.

File

[dsp.h](#)

C

```
int DSP_VectorChkEqu32(int32_t * indata1, int32_t * indata2, int N);
```

Returns

(int) - '1' if vectors are equal, '0' if vectors are not equal

Description

Function DSP_VectorChkEqu32:

```
int DSP_VectorChkEqu32(int32_t *indata1, int32_t *indata2, int N);
```

Compares the first N values of indata1 to the same elements of indata2. The comparison requires that all numbers be in Q31 fractional data format. Returns the integer value '1' if all numbers are equal, and '0' if they are not equal. N must be greater than or equal to four and a multiple of four, or it will be truncated to the nearest multiple of four.

Remarks

None.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Example

```
int outCheck;
int Num = 4;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
```

```

        0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//          1,      -1,      0.9,      0.8,      0.2,      0.5,      1,      -0.6
int32_t inBufTestB[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
        0x80000000, 0x40000000, 0x7FFFFFFF, 0x20000000};
//          1,      -1,      0.9,      0.8,      -1,      0.5,      1,      0.25

outCheck = DSP_VectorChkEqu32(inBufTestA, inBufTestB, Num);

// outCheck = 1 // true for first 4 numbers of series

```

Parameters

Parameters	Description
indata1	pointer to input array 1 of elements (int32_t)
indata2	pointer to input array 2 of elements (int32_t)
N	number of samples (int)

DSP_VectorCopy Function

Copies the elements of one vector to another.

File

[dsp.h](#)

C

```
void DSP_VectorCopy(int32_t * outdata, int32_t * indata, int N);
```

Returns

None.

Description

Function DSP_VectorCopy:

```
void DSP_VectorCopy(int32_t *outdata, int32_t *indata, int N);
```

Fills the first N values of an input vector outdata with the elements from indata. N must be a multiple of four and greater than or equal to four or it will be truncated to the nearest multiple of four. The vector result and the scalar value to fill are both Q31 fractional format.

Remarks

None.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Example

```

int      Num = 4;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
        0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//          1,      -1,      0.9,      0.8,      0.2,      0.5,      1,      -0.6
int32_t inBufTestB[8]={0x00000000, 0x7FFFFFFF, 0x40000000, 0x0CCCCCCC,
        0x40000000, 0x60000000, 0x80000000, 0x20000000};
//          0,      1,      0.5,      0.1,      0.75,      0.5,      -1,      0.25

DSP_VectorCopy(inBufTestA, inBufTestB, Num);

// inBufTestA = {0x00000000, 0x7FFFFFFF, 0x40000000, 0x0CCCCCCC,
//               0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334} // first 4 values copied

```

Parameters

Parameters	Description
outdata	pointer to destination array of values (int32_t)
indata	pointer to source array of elements (int32_t)
N	number of samples (int)

DSP_VectorCopyReverse32 Function

Reverses the order of elements in one vector and copies them into another.

File

[dsp.h](#)

C

```
void DSP_VectorCopyReverse32(int32_t * outdata, int32_t * indata, int N);
```

Returns

None.

Description

Function DSP_VectorCopyReverse32:

```
void DSP_VectorCopyReverse32(int32_t *outdata, int32_t *indata, int N);
```

Fills the first N values of an input vector Outdata with the reverse elements from INDATA. N must be a multiple of 4 and greater than 4 or will be truncated to the nearest multiple of 4. The vectors are both Q31 fractional format.

Remarks

None.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Example

```
int Num = 4;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                      0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//           1,    -1,    0.9,    0.8,    0.2,    0.5,    1,    -0.6
int32_t inBufTestB[8]={0x00000000, 0x7FFFFFFF, 0x40000000, 0x0CCCCCCC,
                      0x40000000, 0x60000000, 0x80000000, 0x20000000};
//           0,    1,    0.5,    0.1,    0.75,    0.5,    -1,    0.25

DSP_VectorCopyReverse32(inBufTestA, inBufTestB, Num);

// inBufTestA = {0x0CCCCCCC, 0x40000000, 0x7FFFFFFF, 0x00000000,
//               0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334}
// first 4 values copied reverse order
```

Parameters

Parameters	Description
outdata	pointer to destination array of values (int32_t)
indata	pointer to source array of elements (int32_t)
N	number of samples (int)

DSP_VectorDivC Function

Divides the first N elements of inVector by a constant divisor, and stores the result in outVector.

File

[dsp.h](#)

C

```
void DSP_VectorDivC(_Q16 * outVector, _Q16 * inVector, _Q16 divisor, int N);
```

Returns

None.

Description

Function DSP_VectorDivC:


```
void DSP_VectorDivC(_Q16 *outVector, _Q16 *inVector, _Q16 divisor, int N);
```

Divides each element of the first N elements of inVector by a constant, divisor. The output is stored to outVector. Both vectors and the scalar are [_Q16](#) format, which is 32-bit data with 15 bits for the integer and 16 bits for the fractional portion. If values exceed maximum or minimum they will saturate to the maximum or minimum respectively.

Remarks

This function uses the Microchip PIC32MZ LibQ library to function. The user must include that library and header file into the design in order to operate this function. For more information on the Div function see the LibQ documentation for [_LIBQ_Q16Div](#).

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. This function uses the Div function from the LibQ library. That library must be compiled as part of the project.

Example

```
int divNum = 4;
_Q16 divScalar = 0x00020000; // 2.0
_Q16 inDivVec[8] = {0x08000000, 0xfffc0000, 0x00024000, 0x00100000, 0x00038000,
                   0x00400000, 0xfffe0000, 0x00058000};
//      2048, -4, 2.25, 16, 3.5, 64, -2, 5.5
_Q16 outDivVec[8] = {0};

DSP_VectorDivC(outDivVec, inDivVec, divScalar, divNum);

// outDivVec = 0x04000000, 0xFFFFE0000, 0x00012000, 0x00080000, 0, 0, 0, 0
//              1024.0,      -2.0,      1.125,      16.0, 0, 0, 0, 0
```

Parameters

Parameters	Description
outdata	pointer to destination array of elements (_Q16)
indata	pointer to source array of elements (_Q16)
divisor	scalar divisor for the input vector (_Q16)
N	number of samples (int)

DSP_VectorDotp16 Function

Computes the dot product of two vectors, and scales the output by a binary factor.

File

[dsp.h](#)

C

```
int16_t DSP_VectorDotp16(int16_t * indata1, int16_t * indata2, int N, int scale);
```

Returns

int16_t - scaled output of calculation, Q15 format

Description

Function DSP_VectorDotp16:

```
int16_t DSP_VectorDotp16(int16_t *indata1, int16_t *indata2, int N, int scale);
```

Calculates the dot product of two input vectors, and scales the output. Function will saturate if it exceeds maximum or minimum values. Scaling is done by binary shifting, after accumulation in a 32 bit register. All calculations are done in Q15 fractional format. return = $1/(2^{\text{scale}}) * \text{sum}(\text{indata1}[i] * \text{indata2}[i])$

Remarks

This must be assembled with .set microMIPS.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Example

```
int16_t inBufMultA[8]={0x7FFF, 0x8000, 0x7333, 0x6666, 0x1999, 0x4000, 0x7FFF, 0xB334};
//              1,      -1,      0.9,      0.8,      0.2,      0.5,      1,      -0.6
int16_t inBufMultB[8]={0x0CCD, 0x0CCD, 0x4000, 0xC000, 0xE667, 0x4000, 0x0000, 0x0CCD};
```

```
//          0.1,    0.1,    0.5,   -0.5,   -0.2,    0.5,    0,    0.1
int Num = 8;
int scaleVal = 2;
int16_t outScalar;

int Num = 8;

outScalar = DSP_VectorDotp16(inBufMultA, inBufMultB, Num, scaleVal);

// outScalar = 1/(2^scaleVal)*(inBufMultA[] dot inBufMultB[]) =
// (1/4) * (0.1 + -0.1 + 0.45 + -0.4 + -0.04 + 0.25 + 0 + -0.06) = 0.25 * 0.20 = 0.05
// = (int16_t)0x0666
```

Parameters

Parameters	Description
indata1	pointer to input array of 16-bit elements (int16_t)
indata2	pointer to input array of 16-bit elements (int16_t)
scale	number of bits to shift return right (int)
N	number of samples (int)

DSP_VectorDotp32 Function

Computes the dot product of two vectors, and scales the output by a binary factor

File

[dsp.h](#)

C

```
int32_t DSP_VectorDotp32(int32_t * indata1, int32_t * indata2, int N, int scale);
```

Returns

int16_t - scaled output of calculation, Q31 format

Description

Function DSP_VectorDotp32:

```
int32_t DSP_VectorDotp32(int32_t *indata1, int32_t *indata2, int N, int scale);
```

Calculates the dot product of two input vectors, and scales the output. Function will saturate if it exceeds maximum or minimum values. Scaling is done by binary shifting, after calculation of the result. All calculations are done in Q31 fractional format. $\text{return} = 1/(2^{\text{scale}}) * \sum(\text{indata1}[i] * \text{indata2}[i])$

Remarks

This must be assembled with .set microMIPS.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Example

```
int32_t inBufMultA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                      0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//          1,    -1,    0.9,    0.8,    0.2,    0.5,    1,    -0.6
int32_t inBufMultB[8]={0x0CCCCCD, 0x0CCCCCD, 0x40000000, 0xC0000000,
                      0xE6666667, 0x40000000, 0x00000000, 0x0CCCCCD};
//          0.1,    0.1,    0.5,   -0.5,   -0.2,    0.5,    0,    0.1
int Num = 8;
int scaleVal = 2;
int32_t outScalar;

int Num = 8;

outScalar = DSP_VectorDotp32(inBufMultA, inBufMultB, Num, scaleVal);

// outScalar = 1/(2^scaleVal)*(inBufMultA[] dot inBufMultB[]) =
// (1/4) * (0.1 + -0.1 + 0.45 + -0.4 + -0.04 + 0.25 + 0 + -0.06) = 0.25 * 0.20 = 0.05
// = (int32_t)0x06666666
```

Parameters

Parameters	Description
indata1	pointer to input array of 16-bit elements (int32_t)
indata2	pointer to input array of 16-bit elements (int32_t)
scale	number of bits to shift return right (int)
N	number of samples (int)

DSP_VectorExp Function

Computes the EXP (e^x) of the first N elements of inVector, and stores the result in outVector.

File

[dsp.h](#)

C

```
void DSP_VectorExp(_Q16 * outVector, _Q16 * inVector, int N);
```

Returns

None.

Description

Function DSP_VectorExp:

```
void DSP_VectorExp(_Q16 *outVector, _Q16 *inVector, int N);
```

Computes the Exp value, e to the power of X, on the first N elements of inVector. The output is stored to outVector. Both vectors are [_Q16](#) format, which is 32-bit data with 15 bits for the integer and 16 bits for the fractional portion. If values exceed maximum or minimum they will saturate to the maximum or zero respectively.

Remarks

Inclusion of the LibQ header file and library is mandatory to use this function.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. This function uses the Exp and Div functions from the LibQ library. That library must be compiled as part of the project.

Example

```
int expNum = 4;
_Q16 inExpVec[8] = {0x00010000, 0xffff0000, 0x00020000, 0x00030000, 0x00038000,
                   0x00040000, 0xfffe0000, 0x00058000};
//      1.0,  -1.0,  2.0,  3.0,  3.5,  4.0,  -2.0,  5.5
_Q16 outExpVec[8] = {0};

DSP_VectorExp(outExpVec, inExpVec, expNum);

// outExpVec = 0x0002B7E1, 0x00005E2D, 0x00076399, 0x001415E6, 0, 0, 0, 0
//            2.71828,  0.26787,   7.3891,  20.0855, 0, 0, 0, 0
```

Parameters

Parameters	Description
outdata	pointer to destination array of elements (_Q16)
indata	pointer to source array of elements (_Q16)
N	number of samples (int)

DSP_VectorFill Function

Fills an input vector with scalar data.

File

[dsp.h](#)

C

```
void DSP_VectorFill(int32_t * indata, int32_t data, int N);
```

Returns

None.

Description

Function DSP_VectorFill:

```
void DSP_VectorFill(int32_t *indata, int32_t data, int N);
```

Fills the first N values of an input vector indata with the value data. N must be a multiple of four and greater than or equal to four or it will be truncated to the nearest multiple of four. The vector result and the scalar value to fill are both Q31 fractional format.

Remarks

None.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Example

```
int32_t fillValue = 0x3FFFFFFF;
int Num = 4;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                      0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//           1,      -1,      0.9,      0.8,      0.2,      0.5,      1,      -0.6

DSP_VectorFill(inBufTestA, fillValue, Num);

// inBufTestA = {0x3FFFFFFF, 0x3FFFFFFF, 0x3FFFFFFF, 0x3FFFFFFF,
//               0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334} // first 4 values filled
```

Parameters

Parameters	Description
indata	pointer to source array of elements (int32_t)
data	scalar value to fill the array (int32_t)
N	number of samples (int)

DSP_VectorLn Function

Computes the Natural Log, Ln(x), of the first N elements of inVector, and stores the result in outVector.

File

[dsp.h](#)

C

```
void DSP_VectorLn(_Q4_11 * outVector, _Q16 * inVector, int N);
```

Returns

None.

Description

Function DSP_VectorLn:

```
void DSP_VectorLn(_Q4_11 *outVector, _Q16 *inVector, int N);
```

Computes the Ln(x) value, on the first N elements of inVector. The output is stored to outVector. Input vector is [_Q16](#) format, which is 32-bit data with 15 bits for the integer and 16 bits for the fractional portion. The output vector is reduced resolution Q4.11 format, which is a 16-bit integer format with 11 bits representing the fractional resolution. If values exceed maximum or minimum they will saturate to the maximum or zero respectively.

Remarks

This function uses the Microchip PIC32MZ LibQ library to function. The user must include that library and header file into the design in order to operate this function. For more information on the Ln function see the LibQ documentation for [_LIBQ_Q4_11_In_Q16](#). A negative number input will return a saturated negative value (0x8000).

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. This function uses the Ln function from the LibQ library. That library must be compiled as part of the project.

Example

```
int lnNum = 4;
_Q16 inLnVal[8] = {0x40000000, 0xffff0000,0x00020000,0x00100000,0x00038000,
                  0x00400000,0xfffe0000,0x00058000};
//      16384.0,  -1.0,  2.0,  16.0,   3.5,   64.0,  -2.0,  5.5
_Q4_11 outLnVal[8] = {0};

DSP_VectorLn(outLnVal, inLnVal, lnNum);

// outLnVal =      0x4DA2,          0x8000,   0x058C,   0x162E,  0,  0,  0,  0
//            9.704,  sat negative,  0.6934,   2.772,  0,  0,  0,  0
```

Parameters

Parameters	Description
outdata	pointer to destination array of elements (_Q16)
indata	pointer to source array of elements (_Q4_11)
N	number of samples (int)

DSP_VectorLog10 Function

Computes the Log10(x), of the first N elements of inVector, and stores the result in outVector.

File

[dsp.h](#)

C

```
void DSP_VectorLog10(_Q3_12 * outVector, _Q16 * inVector, int N);
```

Returns

None.

Description

Function DSP_VectorLog10:

```
void DSP_VectorLog10(_Q3_12 *outVector, _Q16 *inVector, int N);
```

Computes the Log10(x) value, on the first N elements of inVector. The output is stored to outVector. Input vector is [_Q16](#) format, which is 32-bit data with 15 bits for the integer and 16 bits for the fractional portion. The output vector is reduced resolution [Q3.12](#) format, which is a 16-bit integer format with 12 bits representing the fractional resolution. If values exceed maximum or minimum they will saturate to the maximum or zero respectively.

Remarks

This function uses the Microchip PIC32MZ LibQ library to function. The user must include that library and header file into the design in order to operate this function. For more information on the Log10 function see the LibQ documentation for [_LIBQ_Q3_12_log10_Q16](#). A negative number input will return a saturated negative value (0x8000).

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. This function uses the Log10 function from the LibQ library. That library must be compiled as part of the project.

Example

```
int logNum = 4;
_Q16 inLogVal[8] = {0x40000000, 0xffff0000,0x00020000,0x00100000,0x00038000,
                  0x00400000,0xfffe0000,0x00058000};
//      16384.0,  -1.0,  2.0,  16.0,   3.5,   64.0,  -2.0,  5.5
_Q3_12 outLogVal[8] = {0};

DSP_VectorLog10(outLogVal, inLogVal, logNum);

// outLogVal =      0x436E,          0x8000,   0x04D1,   0x1344,  0,  0,  0,  0
```

```
//          4.2144,  sat negative,  0.3010,  1.2041,  0,  0,  0,  0
```

Parameters

Parameters	Description
outdata	pointer to destination array of elements (_Q16)
indata	pointer to source array of elements (_Q3_12)
N	number of samples (int)

DSP_VectorLog2 Function

Computes the Log2(x) of the first N elements of inVector, and stores the result in outVector.

File

[dsp.h](#)

C

```
void DSP_VectorLog2(_Q5_10 * outVector, _Q16 * inVector, int N);
```

Returns

None.

Description

Function DSP_VectorLog2:

```
void DSP_VectorLog2(_Q5_10 *outVector, _Q16 *inVector, int N);
```

Computes the Log2 value, where $\log_2(x) = \ln(x) * \log_2(e)$, on the first N elements of inVector. The output is stored to outVector. Input vector is [_Q16](#) format, which is 32-bit data with 15 bits for the integer and 16 bits for the fractional portion. The output vector is reduced resolution [Q5.10](#) format, which is a 16-bit integer format with 10 bits representing the fractional resolution. If values exceed maximum or minimum they will saturate to the maximum or zero respectively.

Remarks

This function uses the Microchip PIC32MZ LibQ library to function. The user must include that library and header file into the design in order to operate this function. For more information on the Log2 function see the LibQ documentation for [_LIBQ_Q5_10_log2_Q16](#). A negative number input will return a saturated negative value (0x8000).

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. This function uses the Log2 function from the LibQ library. That library must be compiled as part of the project.

Example

```
int log2Num = 4;
_Q16 inLog2Val[8] = {0x40000000, 0xffff0000, 0x00020000, 0x00030000, 0x00038000,
                    0x00040000, 0xfffe0000, 0x00058000};
//          16384.0,  -1.0,  2.0,  3.0,  3.5,  4.0,  -2.0,  5.5
_Q5_10 outLog2Val[8] = {0};

DSP_VectorLog2(outLog2Val, inLog2Val, log2Num);

// outLog2Val =  0x3800,      0x8000,   0x0400,   0x0657,  0,  0,  0,  0
//              14.0,  sat negative,   1.0,   1.585,  0,  0,  0,  0
```

Parameters

Parameters	Description
outdata	pointer to destination array of elements (_Q16)
indata	pointer to source array of elements (_Q5_10)
N	number of samples (int)

DSP_VectorMax32 Function

Returns the maximum value of a vector.

File

[dsp.h](#)

C

```
int32_t DSP_VectorMax32(int32_t * indata, int N);
```

Returns

(int32_t) - maximum value within the vector, Q31 format

Description

Function DSP_VectorMax32:

```
int32_t DSP_VectorMax32(int32_t *indata, int N);
```

Returns the highest value of the first N elements of the vector indata. The comparison requires that all numbers be in Q31 fractional data format. N must be greater than or equal to four and a multiple of four, or it will be truncated to the nearest multiple of four.

Remarks

None.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and multiple of four.

Example

```
int32_t  outCheck;
int      Num = 4;
int32_t  inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                       0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//          1,      -1,      0.9,    0.8,    0.2,    0.5,    1,      -0.6

outCheck = DSP_VectorMax32(inBufTestA, Num);

// outCheck = 0x7FFFFFFF // first 4 values
```

Parameters

Parameters	Description
indata	pointer to input array of elements (int32_t)
N	number of samples (int)

DSP_VectorMaxIndex32 Function

Returns the index of the maximum value of a vector.

File

[dsp.h](#)

C

```
int DSP_VectorMaxIndex32(int32_t * indata, int N);
```

Returns

int - index of the position of the maximum array element

Description

Function DSP_VectorMaxIndex32:

```
int DSP_VectorMaxIndex32(int32_t *indata, int N);
```

Returns the index of the highest value of the first N elements of the vector indata. The comparison requires that all numbers be in Q31 fractional data format. N must be greater than or equal to four and a multiple of four, or it will be truncated to the nearest multiple of four.

Remarks

Index values range from 0 .. (n-1).

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Example

```
int    indexValue;
int    Num = 8;
int32_t inBufTestA[8]={0x3FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                      0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//          0.5,    -1,    0.9,    0.8,    0.2,    0.5,    1,    -0.6

indexValue = DSP_VectorMaxIndex32(inBufTestA, Num);

// returnValue = 6 (position corresponding to 0x7FFFFFFF)
```

Parameters

Parameters	Description
indata	pointer to source array of elements (int32_t)
N	number of samples (int)

DSP_VectorMean32 Function

Calculates the mean average of an input vector.

File

[dsp.h](#)

C

```
int32_t DSP_VectorMean32(int32_t * indata1, int N);
```

Returns

int32_t - mean average value of the vector

Description

Function DSP_VectorMean32:

```
int32_t DSP_VectorMean32(int32_t *indata, int N);
```

Calculates the mean average of the first N elements of the vector indata. The values of indata1 are in Q31 fractional format. The value N must be greater than or equal to four and a multiple of four, or it will be truncated to the nearest multiple of four.

Remarks

None.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Example

```
int32_t    returnValue;
int    Num = 4;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                      0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//          1,    -1,    0.9,    0.8,    0.2,    0.5,    1,    -0.6

returnValue = DSP_VectorMean32(inBufTestA, Num);

// returnValue = 0x36666666 = (1-1+0.9+0.8)/4 = 0.425
```

Parameters

Parameters	Description
indata	pointer to source array of elements (int32_t)
N	number of samples (int)

DSP_VectorMin32 Function

Returns the minimum value of a vector.

File

[dsp.h](#)

C

```
int32_t DSP_VectorMin32(int32_t * input, int N);
```

Returns

(int32_t) - minimum value within the vector, Q31 format

Description

Function DSP_VectorMin32:

```
int32_t DSP_VectorMin32(int32_t *indata, int N);
```

Returns the lowest value of the first N elements of the vector indata. The comparison requires that all numbers be in Q31 fractional data format. N must be greater than or equal to four and a multiple of four, or it will be truncated to the nearest multiple of four.

Remarks

None.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and multiple of four.

Example

```
int32_t outCheck;
int Num = 4;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                      0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//          1,      -1,      0.9,    0.8,    0.2,    0.5,    1,      -0.6

outCheck = DSP_VectorMin32(inBufTestA, Num);

// outCheck = 0x80000000 // first 4 values
```

Parameters

Parameters	Description
indata	pointer to input array of elements (int32_t)
N	number of samples (int)

DSP_VectorMinIndex32 Function

Returns the index of the minimum value of a vector.

File

[dsp.h](#)

C

```
int DSP_VectorMinIndex32(int32_t * indata, int N);
```

Returns

int32_t - mean average value of the vector

Description

Function DSP_VectorMinIndex32:

```
int DSP_VectorMinIndex32(int32_t *indata, int N);
```

Returns the relative position index of the lowest value of the first N elements of the vector indata. The comparison requires that all numbers be in Q31 fractional data format. N must be greater than or equal to four and a multiple of four, or it will be truncated to the nearest multiple of four.

Remarks

Index values range from 0 .. (n-1).

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Example

```
int    indexValue;
int    Num = 8;
int32_t inBufTestA[8]={0x3FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                      0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//          0.5,    -1,    0.9,    0.8,    0.2,    0.5,    1,    -0.6

indexValue = DSP_VectorMinIndex32(inBufTestA, Num);

// returnValue = 1 (position corresponding to 0x80000000)
```

Parameters

Parameters	Description
indata	pointer to source array of elements (int32_t)
N	number of samples (int)

DSP_VectorMul16 Function

Multiplication of a series of numbers in one vector to another vector.

File

dsp.h

C

```
void DSP_VectorMul16(int16_t * outdata, int16_t * indata1, int16_t * indata2, int N);
```

Returns

None.

Description

Function DSP_VectorMul16:

```
void DSP_VectorMul16(int16_t *outdata, int16_t *indata1, int16_t *indata2, int N);
```

Multiplies the value of each element of indata1 * indata2 and stores it to outdata. The number of samples to process is given by the parameter N. Data is in the Q15 fractional format. outdata[i] filled with N elements of indata1[i] * indata2[i]

Remarks

This must be assembled with .set microMIPS.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Example

```
int16_t *pOutdata;
int16_t outVal[8];
int16_t inBufMultA[8]={0x7FFF, 0x8000, 0x7333, 0x6666, 0x1999, 0x4000, 0x7FFF, 0xB334};
//          1,    -1,    0.9,    0.8,    0.2,    0.5,    1,    -0.6
int16_t inBufMultB[8]={0x0CCD, 0x4000, 0x0000, 0xE667, 0x4000, 0x0000, 0x0CCD};
//          0.1,    0.1,    0.5,    -0.5,    -0.2,    0.5,    0,    0.1
int Num = 8;

pOutdata = &outVal;

DSP_VectorMul16(pOutdata, inBufMultA, inBufMultB, Num);

// outVal[i] = inBufTest[i] * inBuf2[i] =
// {0x0CCD, 0xF333, 0x399A, 0xCCCC, 0xFAE2, 0x2000, 0x0000, 0xF852}
// 0.1, -0.1, 0.45, -0.4, -0.04, 0.25, 0, -0.06
```

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int16_t)
indata1	pointer to input array of 16-bit elements (int16_t)

indata2	pointer to input array of 16-bit elements (int16_t)
N	number of samples (int)

DSP_VectorMul32 Function

Multiplication of a series of numbers in one vector to another vector.

File

[dsp.h](#)

C

```
void DSP_VectorMul32(int32_t * outdata, int32_t * indata1, int32_t * indata2, int N);
```

Returns

None.

Description

Function DSP_VectorMul32:

```
void DSP_VectorMul32(int32_t *outdata, int32_t *indata1, int32_t *indata2, int N);
```

Multiples the value of each element of indata1 * indata2 and stores it to outdata. The number of samples to process is given by the parameter N. Data is in the Q31 fractional format. outdata[i] filled with N elements of indata1[i] * indata2[i]

Remarks

This must be assembled with .set microMIPS.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Example

```
int16_t *pOutdata;
int32_t outVal[8];
int32_t inBufMultA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                      0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//          1,    -1,    0.9,    0.8,    0.2,    0.5,    1,    -0.6
int32_t inBufMultB[8]={0x0CCCCCD, 0x0CCCCCD, 0x40000000, 0xC0000000,
                      0xE6666667, 0x40000000, 0x00000000, 0x0CCCCCD};
//          0.1,    0.1,    0.5,    -0.5,    -0.2,    0.5,    0,    0.1
int Num = 8;

pOutdata = &outVal;

DSP_VectorMul32(pOutdata, inBufMultA, inBufMultB, Num);

// outVal[i] = inBufTest[i] * inBuf2[i] =
// {0x0CCCCCD, 0xF3333333, 0x3999999A, 0xCCCCCD, 0xFAE147AE,
//   0x20000000, 0x00000000, 0xF851EB86}
// 0.1, -0.1, 0.45, -0.4, -0.04, 0.25, 0, -0.06
```

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int32_t)
indata1	pointer to input array of 16-bit elements (int32_t)
indata2	pointer to input array of 16-bit elements (int32_t)
N	number of samples (int)

DSP_VectorMulc16 Function

Multiplication of a series of numbers in one vector to a scalar value.

File

[dsp.h](#)

C

```
void DSP_VectorMulc16(int16_t * outdata, int16_t * indata, int16_t c, int N);
```

Returns

None.

Description

Function DSP_VectorMulc16:

```
void DSP_VectorMulc16(int16_t *outdata, int16_t *indata, int16_t c, int N);
```

Multiplies the value of each element of indata1 * c and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q15 fractional format. outdata[i] filled with N elements of indata[i] * c

Remarks

This must be assembled with .set microMIPS.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Example

```
int16_t *pOutdata;
int16_t outVal[8];
int16_t inBufMultA[8]={0x7FFF, 0x8000, 0x7333, 0x6666, 0x1999, 0x4000, 0x0000, 0xB334};
//                               1,    -1,    0.9,    0.8,    0.2,    0.5,    0,    -0.6
int16_t constValue = 0x4000;
int Num = 8;

pOutdata = &outVal;

DSP_VectorMulc16(pOutdata, inBufMultA, constValue, Num);

// outVal[i] = inBufTest[i] * constValue =
// {0x4000, 0xC000, 0x399A, 0x3333, 0x1999, 0x2000, 0x0000, 0xD99A}
// 0.5, -0.5, 0.45, 0.4, 0.1, 0.25, 0, -0.3
```

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int16_t)
indata	pointer to input array of 16-bit elements (int16_t)
c	scalar multiplicand (int16_t)
N	number of samples (int)

DSP_VectorMulc32 Function

Multiplication of a series of numbers in one vector to a scalar value.

File

[dsp.h](#)

C

```
void DSP_VectorMulc32(int32_t * outdata, int32_t * indata, int32_t c, int N);
```

Returns

None.

Description

Function DSP_VectorMulc32:

```
void DSP_VectorMulc32(int32_t *outdata, int32_t *indata, int32_t c, int N);
```

Multiplies the value of each element of indata * c and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q31 fractional format. outdata[i] filled with N elements of indata[i] * c

Remarks

This must be assembled with `.set microMIPS`.

Preconditions

The pointers `outdata` and `indata` must be aligned on 4-byte boundaries. `N` must be greater than or equal to four and a multiple of four.

Example

```
int16_t *pOutdata;
int32_t outVal[8];
int32_t inBufMultA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                      0x19999999, 0x40000000, 0x00000000, 0xB3333334};
//          1,   -1,   0.9,   0.8,   0.2,   0.5,   1,   -0.6
int32_t constValue = 0x4000;
int Num = 8;

pOutdata = &outVal;

DSP_VectorMulc32(pOutdata, inBufMultA, constValue, Num);

// outVal[i] = inBufTest[i] * constValue =
// {0x40000000, 0xC0000000, 0x3999999A, 0x33333333, 0x19999999,
//   0x20000000, 0x00000000, 0xD999999A}
//   0.5,   -0.5,   0.45,   0.4,   0.1,   0.25,   0,   -0.3
```

Parameters

Parameters	Description
<code>outdata</code>	pointer to output array of 16-bit elements (<code>int32_t</code>)
<code>indata</code>	pointer to input array of 16-bit elements (<code>int32_t</code>)
<code>c</code>	scalar multiplicand (<code>int32_t</code>)
<code>N</code>	number of samples (<code>int</code>)

DSP_VectorNegate Function

Inverses the sign (negates) the elements of a vector.

File

[dsp.h](#)

C

```
void DSP_VectorNegate(int32_t * outdata, int32_t * indata, int N);
```

Returns

None.

Description

Function `DSP_VectorNegate`:

```
void DSP_VectorNegate(int32_t *outdata, int32_t *indata, int N);
```

Sign inversion of the first `N` values of an `indata` are assigned to `outdata`. `N` must be a multiple of four and greater than or equal to four or it will be truncated to the nearest multiple of four. The vector result and the scalar value to fill are both Q31 fractional format.

Remarks

None.

Preconditions

The pointers `outdata` and `indata` must be aligned on 4-byte boundaries. `N` must be greater than or equal to four and a multiple of four.

Example

```
int Num = 4;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                      0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//          1,   -1,   0.9,   0.8,   0.2,   0.5,   1,   -0.6
int32_t outBufTest[8]={0x0CCCCCD, 0x0CCCCCD, 0x40000000, 0xC0000000,
```

```

                                0xE6666667, 0x40000000, 0x00000000, 0x0CCCCCD};
//                                0.1,    0.1,    0.5,    -0.5,    -0.2,    0.5,    0,    0.1;

DSP_VectorNegate(outBufTest, inBufTestA, Num);

// inBufTestA = {0x80000000, 0x7FFFFFFF, 0x8CCCCCD, 0x9999999A,
//               0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334} // first 4 values neg
//               -1,    1,    -0.9,    -0.8,    -0.2,    0.5,    0,    0.1

```

Parameters

Parameters	Description
outdata	pointer to destination array of elements (int32_t)
indata	pointer to source array of elements (int32_t)
N	number of samples (int)

DSP_VectorRecip Function

Computes the reciprocal (1/x) of the first N elements of inVector, and stores the result in outVector.

File

dsp.h

C

```
void DSP_VectorRecip(_Q16 * outVector, _Q16 * inVector, int N);
```

Returns

None.

Description

Function DSP_VectorRecip:

```
void DSP_VectorRecip(_Q16 *outVector, _Q16 *inVector, int N);
```

Computes the reciprocal (1/x) on the first N elements of inVector. The output is stored to outVector. Both vectors are Q16 format, which is 32-bit data with 15 bits for the integer and 16 bits for the fractional portion. If values exceed maximum or minimum they will saturate to the maximum or minimum respectively.

Remarks

This function uses the Microchip PIC32MZ LibQ library to function. The user must include that library and header file into the design in order to operate this function. For more information on the Div function see the LibQ documentation for [_LIBQ_Q16Div](#). A value of zero in the array will not cause an error, but will return 0.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. This function uses the Div function from the LibQ library. That library must be compiled as part of the project.

Example

```

int recNum = 4;
_Q16 inRecVec[8] = {0x08000000, 0xfffc0000, 0x00020000, 0x00100000, 0x00038000,
                  0x00400000, 0xfffe0000, 0x00058000};
//                2048.0, -4.0, 2.0, 16.0, 3.5, 64.0, -2.0, 5.5
_Q16 outRecVec[8] = {0};

DSP_VectorRecip(outRecVec, inRecVec, recNum);

// outRecVec = 0x00000020, 0xFFFFC000, 0x00008000, 0x00001000, 0, 0, 0, 0
//                0.000488,    -0.25,    0.5,    0.0625, 0, 0, 0, 0

```

Parameters

Parameters	Description
outdata	pointer to destination array of elements (<u>Q16</u>)
indata	pointer to source array of elements (<u>Q16</u>)
N	number of samples (int)

DSP_VectorRMS16 Function

Computes the root mean square (RMS) value of a vector.

File

[dsp.h](#)

C

```
int16_t DSP_VectorRMS16(int16_t * inVector, int N);
```

Returns

int16_t - RMS function output, Q15 format

Description

Function DSP_VectorRMS16:

```
int16_t DSP_VectorRMS16(int16_t *inVector, int N);
```

Computes the root mean square value of the first N values of inVector. Both input and output are Q15 fractional values. The function will saturate if maximum or minimum values are exceeded.

Remarks

This function is optimized with microMIPS and M14KCe ASE DSP instructions. This function is dependent on the LibQ library, and uses the [_LIBQ_Q16Sqrt](#) external function call.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and multiple of four.

Example

```
int16_t vecRMSIn[32]={0x1999, 0xD99A, 0x4000, 0x2666,0x1999,0x1999,0x2666, 0x3333};
// 0.2, -0.3, 0.5, 0.3, 0.2, 0.2, 0.3, 0.4
int16_t RMSOut=0;
int Nrms = 8;

RMSOut = DSP_VectorRMS16(vecRMSIn, Nrms);

// RMSOut = 0x287C (= 0.31628)
```

Parameters

Parameters	Description
indata	pointer to input array of 16-bit elements (int16_t)
N	number of samples (int)

DSP_VectorShift Function

Shifts the data index of an input data vector.

File

[dsp.h](#)

C

```
void DSP_VectorShift(int32_t * outdata, int32_t * indata, int N, int shift);
```

Returns

None.

Description

Function DSP_VectorShift:

```
void DSP_VectorShift(int32_t *outdata, int32_t *indata, int N, int shift);
```

Shifts N data elements of indata to outdata, with an index change of shift. The amount of data shifted includes zero padding for the first (shift) elements if shift is positive. The vector size of indata and outdata need not be the same, however, N must not exceed either array size.

Remarks

Destination array values shift to left (relative to the input vector) when shift is positive (back filled with zeros) and shift to the right when shift is negative. The total amount of values copied to the destination array is the length of N less the shift amount.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must not exceed the amount of elements in the source array. shift must not exceed the number of elements in the destination array.

Example

```
int shiftValue = 3;
int Num = 8;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                      0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//          1,      -1,      0.9,    0.8,    0.2,    0.5,    1,      -0.6
int32_t inBufTestB[8]={0x80000000, 0x7FFFFFFF, 0x40000000, 0x0CCCCCCC,
                      0x40000000, 0x60000000, 0x80000000, 0x20000000};
//          -1,     1,      0.5,    0.1,    0.75,   0.5,   -1,     0.25

DSP_VectorShift(inBufTestA, inBufTestB, Num, shiftValue);

// inBufTestA = {0x00000000, 0x00000000, 0x00000000, 0x80000000,
//               0x7FFFFFFF, 0x40000000, 0x0CCCCCCC, 0x40000000} // shifted 3 positive
```

Parameters

Parameters	Description
outdata	pointer to destination array of elements (int32_t)
indata	pointer to source array of elements (int32_t)
N	number of samples (int)
shift	number of indexes to shift (int)

DSP_VectorSqrt Function

Computes the square root of the first N elements of inVector, and stores the result in outVector.

File

[dsp.h](#)

C

```
void DSP_VectorSqrt(_Q16 * outVector, _Q16 * inVector, int N);
```

Returns

None.

Description

Function DSP_VectorSqrt:

```
void DSP_VectorSqrt(_Q16 *outVector, _Q16 *inVector, int N);
```

Computes the Sqrt(x) on the first N elements of inVector. The output is stored to outVector. Both vectors are Q16 format, which is 32-bit data with 15 bits for the integer and 16 bits for the fractional portion. If values exceed maximum or minimum they will saturate to the maximum or zero respectively.

Remarks

This function uses the Microchip PIC32MZ LibQ library to function. The user must include that library and header file into the design in order to operate this function. For more information on the Sqrt function see the LibQ documentation for [_LIBQ_Q16Sqrt](#). A negative number input will return a saturated value (0x00FFFFxx).

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. This function uses the Sqrt function from the LibQ library. That library must be compiled as part of the project.

Example

```
int sqrtNum = 4;
_Q16 inSqrtVec[8] = {0x40000000, 0xffff0000, 0x00020000, 0x00100000, 0x00038000,
```



```

        0x00400000,0xfffe0000,0x00058000};
//      16384.0, -1.0, 2.0, 16.0, 3.5, 64.0, -2.0, 5.5
_Q16 outSqrtVec[8] = {0};

DSP_VectorSqrt(outSqrtVec, inSqrtVec, sqrtNum);

// outSqrtVec = 0x00800000, 0x00FFF80, 0x00016A0A, 0x00040000, 0, 0, 0, 0
//      128.0, sat negative, 1.41422, 4.0, 0, 0, 0, 0

```

Parameters

Parameters	Description
outdata	pointer to destination array of elements (<u>Q16</u>)
indata	pointer to source array of elements (<u>Q16</u>)
N	number of samples (int)

DSP_VectorStdDev16 Function

Computes the Standard Deviation of a Vector.

File

[dsp.h](#)

C

```
int16_t DSP_VectorStdDev16(int16_t * inVector, int N);
```

Returns

int16_t - Standard Deviation of N selected elements

Description

Function DSP_VectorStdDev16:

```
int16_t DSP_VectorStdDev16(int16_t *inVector, int N);
```

Calculates the standard deviation on the first N elements of inVector and returns the 16-bit scalar result. The standard deviation is the square root of the variance, which is a measure of the delta from mean values. The mean value of the vector is computed in the process. The function has the form -

$StdDev = \sqrt{\sum_{i=0}^{N-1} (x(i) - M(N))^2 / (N-1)}$ where N is the number of vector elements x(i) is a single element in the vector M(N) is the mean of the N elements of the vector

Input values of the vector and output scalar value is Q15 fractional format. This format has data that ranges from -1 to 1, and has internal saturation limits of those same values. Some care has been taken to reduce the impact of saturation by adding processing steps to effectively complete the processing in blocks. However, in some extreme cases of data variance it is still possible to reach the saturation limits.

Remarks

The input vector elements number, N, must be at least 4 and a multiple of 4. This function is optimized with microMIPS and M14KCe ASE DSP instructions. This function is dependent on the LibQ library, and the LIBQ_Q16Sqrt specifically.

Preconditions

The pointers inVector must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four. Dependent on use of the LibQ library.

Example

```

int16_t vecStDevIn[32]={0x4000, 0xD99A, 0x1000, 0x6000,0x1999,0x1999,0x2666, 0x3333};
//      .2, -.3, .125, .75, .2, .2, .3, .4
int16_t StDevOut, Var16Out;
int Nstdev = 4;

StDevOut = DSP_VectorStdDev16(vecStDevIn, Nstdev);
// StDevOut = 0x3A9E (= 0.45797)

```

Parameters

Parameters	Description
inVector	pointer to source array of elements (int16_t)
N	number of samples (int)

DSP_VectorSub16 Function

Calculate the difference of two vectors.

File

[dsp.h](#)

C

```
void DSP_VectorSub16(int16_t * outdata, int16_t * indata1, int16_t * indata2, int N);
```

Returns

None.

Description

Function DSP_VectorSub16:

```
void DSP_VectorSub16(int16_t *outdata, int16_t *indata1, int16_t *indata2, int N);
```

Computes the difference value of each element of indata1 - indata2 and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q15 fractional format. outdata[i] filled with N elements of indata1[i] - indata2[i]

Remarks

This must be assembled with .set microMIPS.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Example

```
int16_t *pOutdata;
int16_t outVal[8];
int16_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100, 200, 127,-127,-2};
int16_t inBuf2[16]=    { 1,2, 3,4, 5,6, 7, 8, 9, 10,-1,-100,-127,127,-7, 0};
int Num = 8;

pOutdata = &outVal;

DSP_VectorSub16(pOutdata, inBufTest, inBuf2, Num);

// outVal[i] = inBufTest[i] - inBuf2[i] = {-6,0,-6,0,-6,-6,-9,-16}
```

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int16_t)
indata1	pointer to input array of 16-bit elements (int16_t)
indata2	pointer to input array of 16-bit elements (int16_t)
N	number of samples (int)

DSP_VectorSub32 Function

Calculate the difference of two vectors.

File

[dsp.h](#)

C

```
void DSP_VectorSub32(int32_t * outdata, int32_t * indata1, int32_t * indata2, int N);
```

Returns

None.

Description

Function DSP_VectorSub32:

```
void DSP_VectorSub32(int32_t *outdata, int32_t *indata1, int32_t *indata2, int N);
```

Computes the difference value of each element of `indata1` - `indata2` and stores it to `outdata`. The number of samples to process is given by the parameter `N`. Data is in a Q31 fractional format. `outdata[i]` filled with `N` elements of `indata1[i] - indata2[i]`

Remarks

This must be assembled with `.set microMIPS`.

Preconditions

The pointers `outdata` and `indata` must be aligned on 4-byte boundaries. `N` must be greater than or equal to four and a multiple of four.

Example

```
int16_t *pOutdata;
int32_t outVal[8];
int32_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100, 200, 127,-127,-2};
int32_t inBuf2[16]=    { 1,2, 3,4, 5,6, 7, 8, 9, 10,-1,-100,-127,127,-7, 0};
int Num = 8;

pOutdata = &outVal;

DSP_VectorSub32(pOutdata, inBufTest, inBuf2, Num);

// outVal[i] = inBufTest[i] - inBuf2[i] = {-6,0,-6,0,-6,-6,-9,-16}
```

Parameters

Parameters	Description
<code>outdata</code>	pointer to output array of 16-bit elements (<code>int32_t</code>)
<code>indata1</code>	pointer to input array of 16-bit elements (<code>int32_t</code>)
<code>indata2</code>	pointer to input array of 16-bit elements (<code>int32_t</code>)
<code>N</code>	number of samples (<code>int</code>)

DSP_VectorSumSquares16 Function

Computes the sum of squares of a vector, and scales the output by a binary factor.

File

`dsp.h`

C

```
int16_t DSP_VectorSumSquares16(int16_t * indata, int N, int scale);
```

Returns

`int16_t` - scaled output of calculation, Q15 format

Description

Function `DSP_VectorSumSquares16`:

```
int16_t DSP_VectorSumSquares16(int16_t *indata, int N, int scale);
```

Calculates the sum of the squares of each element of an input vector, and scales the output. Function will saturate if it exceeds maximum or minimum values. Scaling is done by binary shifting, after accumulation in a 32 bit register. All calculations are done in Q15 fractional format. return = $1/(2^{\text{scale}}) * \text{sum}(\text{indata}[i]^2)$

Remarks

This must be assembled with `.set microMIPS`.

Preconditions

The pointers `outdata` and `indata` must be aligned on 4-byte boundaries. `N` must be greater than or equal to eight and a multiple of eight.

Example

```
int16_t inBufMultA[8]={0x7FFF, 0x8000, 0x7333, 0x6666, 0x1999, 0x4000, 0x7FFF, 0xB334};
//                               1,    -1,    0.9,    0.8,    0.2,    0.5,    1,    -0.6
int Num = 8;
int scaleVal = 3;
int16_t outScalar;

outScalar = DSP_VectorSumSquares16(inBufMultA, Num, scaleVal);
```

```
// outScalar = 1/(2^scaleVal)* sum(inBufMultA[i]^2) =
// (1/8) * (1 + 1 + 0.81 + 0.64 + 0.04 + 0.25 + 1 + 0.36) = 0.125 * 5.1 = 0.6375
// = (int16_t)0x5199
```

Parameters

Parameters	Description
indata	pointer to input array of 16-bit elements (int16_t)
scale	number of bits to shift return right (int)
N	number of samples (int)

DSP_VectorSumSquares32 Function

Computes the sum of squares of a vector, and scales the output by a binary factor.

File

[dsp.h](#)

C

```
int32_t DSP_VectorSumSquares32(int32_t * indata, int N, int scale);
```

Returns

int32_t - scaled output of calculation, Q15 format

Description

Function DSP_VectorSumSquares32:

```
int32_t DSP_VectorSumSquares32(int32_t *indata, int N, int scale);
```

Calculates the sum of the squares of each element of an input vector, and scales the output. The function will saturate if it exceeds maximum or minimum values. Scaling is done by binary shifting, after calculation of the results. All calculations are done in Q31 fractional format. return = $1/(2^{\text{scale}}) * \text{sum}(\text{indata}[i]^2)$

Remarks

This must be assembled with .set microMIPS.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and multiple of four.

Example

```
int32_t inBufMultA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                     0x19999999, 0x40000000, 0x00000000, 0xB3333334};
//          1,    -1,    0.9,    0.8,    0.2,    0.5,    1,    -0.6
int Num = 8;
int scaleVal = 3;
int32_t outScalar;

outScalar = DSP_VectorSumSquares32(inBufMultA, Num, scaleVal);

// outScalar = 1/(2^scaleVal)* sum(inBufMultA[i]^2) =
// (1/8) * (1 + 1 + 0.81 + 0.64 + 0.04 + 0.25 + 1 + 0.36) = 0.125 * 5.1 = 0.6375
// = (int32_t)0x51999999
```

Parameters

Parameters	Description
indata	pointer to input array of 16-bit elements (int32_t)
scale	number of bits to shift return right (int)
N	number of samples (int)

DSP_VectorVari16 Function

Computes the variance of N elements of a Vector.

File

dsp.h

C

```
int16_t DSP_VectorVari16(int16_t * inVector, int N);
```

Returns

int16_t - Variance of N selected elements

Description

Function DSP_VectorVari16:

```
int16_t DSP_VectorVari16(int16_t *inVector, int N);
```

Calculates the variance on the first N elements of inVector and returns the 16-bit scalar result. The variance is a measure of the delta from mean values, and the mean value of the vector is computed in the process. The function has the form -

$var = \sum_{i=0}^{N-1} ((x(i) - M(N))^2) / (N-1)$ where N is the number of vector elements x(i) is a single element in the vector M(N) is the mean of the N elements of the vector

Input values of the vector and output scalar value is Q15 fractional format. This format has data that ranges from -1 to 1, and has internal saturation limits of those same values. Some care has been taken to reduce the impact of saturation by adding processing steps to effectively complete the processing in blocks. However, in some extreme cases of data variance it is still possible to reach the saturation limits.

Remarks

The input vector elements number, N, must be at least 4 and a multiple of 4. This function is optimized with microMIPS and M14KCe ASE DSP instructions.

Preconditions

The pointers inVector must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Example

```
int16_t vecStDevIn[32]={0x4000, 0xD99A, 0x1000, 0x6000,0x1999,0x1999,0x2666, 0x3333};
// .2, -.3, .125, .75, .2, .2, .3, .4
int16_t Var16Out;
int Nvar = 4;

Var16Out= DSP_VectorVari16(vecStDevIn, Nvar); // 16-bit variance function
// Var16Out = 0x1AD8 (= 0.20974)
```

Parameters

Parameters	Description
inVector	pointer to source array of elements (int16_t)
N	number of samples (int)

DSP_VectorVariance Function

Computes the variance of N elements of inVector.

File

dsp.h

C

```
int32_t DSP_VectorVariance(int32_t * inVector, int N);
```

Returns

int32_t - Variance of N selected elements

Description

Function DSP_VectorVariance:

```
int32_t DSP_VectorVariance(int32_t *inVector, int N);
```

Calculates the variance on the first N elements of inVector and returns the 32-bit scalar result. The variance is a measure of the delta from mean values, and the mean value of the vector is computed in the process. The function has the form -

$var = \sum_{i=0}^{N-1} ((x(i) - M(N))^2) / (N-1)$ where N is the number of vector elements x(i) is a single element in the vector M(N) is the mean of the N elements of the vector

Input values of the vector and output scalar value is Q31 fractional format. This format has data that ranges from -1 to 1, and has internal saturation limits of those same values. Some care has been taken to reduce the impact of saturation by adding processing steps to effectively complete the processing in blocks. However, in some extreme cases of data variance it is still possible to reach the saturation limits.

Remarks

The input vector elements number, N, must be at least 4 and a multiple of 4. This function is optimized with microMIPS and M14KCe ASE DSP instructions.

Preconditions

The pointers inVector must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Example

```
int varN = 8;

int32_t inVarVec[8] = {0xE6666667, 0x40000000, 0x40000000, 0x0CCCCCCC,
                     0x00000000, 0x59999999, 0x20000000, 0xC0000000};
//      -0.2, 0.5, 0.5, 0.1, 0, 0.7, 0.25, -0.5
int32_t outVar = 0;

outVar = DSP_VectorVariance(inVarVec, varN);

// outVar == 0x1490D2A6, = 0.1606696
```

Parameters

Parameters	Description
inVector	pointer to source array of elements (int32_t)
N	number of samples (int)

DSP_VectorZeroPad Function

Fills an input vector with zeros.

File

dsp.h

C

```
void DSP_VectorZeroPad(int32_t * indata, int N);
```

Returns

None.

Description

Function DSP_VectorZeroPad:

```
void DSP_VectorZeroPad(int32_t *indata, int N);
```

Fills the first N values of an input vector indata with the value zero. N must be a multiple of four and greater than or equal to four or it will be truncated to the nearest multiple of four. The vector result is in Q31 fractional format.

Remarks

None.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Example

```
int Num = 4;
int32_t inBufTestA[8]={0x3FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                     0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//      0.5, -1, 0.9, 0.8, 0.2, 0.5, 1, -0.6

DSP_VectorZeroPad(inBufTestA, Num);

// inBufTestA = {0x00000000, 0x00000000, 0x00000000, 0x00000000,
//               0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
```

```
//          0,    0,    0,    0,    0.2,  0.5,  1,    -0.6
```

Parameters

Parameters	Description
indata	pointer to source array of elements (int32_t)
N	number of samples (int)

f) Support Functions

mul16 Function

File

[dsp.h](#)

C

```
static inline int16_t mul16(int16_t a, int16_t b);
```

Description

multiply and shift integer

mul16r Function

File

[dsp.h](#)

C

```
static inline int16_t mul16r(int16_t a, int16_t b);
```

Description

multiply and shift Q15

mul32 Function

File

[dsp.h](#)

C

```
static inline int32_t mul32(int32_t a, int32_t b);
```

Description

multiply and shift Q31

SAT16 Function

File

[dsp.h](#)

C

```
static inline int32_t SAT16(int32_t x);
```

Description

saturate both positive and negative Q15

SAT16N Function

File

[dsp.h](#)

C

```
static inline int32_t SAT16N(int32_t x);
```

Description

saturate negative Q15

SAT16P Function**File**

dsp.h

C

```
static inline int32_t SAT16P(int32_t x);
```

Description

saturate positive Q15

g) Data Types and Constants**biquad16 Structure****File**

dsp.h

C

```
typedef struct {
    int16_t a1;
    int16_t a2;
    int16_t b1;
    int16_t b2;
} biquad16;
```

Members

Members	Description
int16_t a1;	feedback delay 1 coef
int16_t a2;	feedback delay 2 coef
int16_t b1;	feedforward delay 1 coef
int16_t b2;	feedforward delay 1 coef

Description

Q15 biquad

int16c Structure**File**

dsp.h

C

```
typedef struct {
    int16_t re;
    int16_t im;
} int16c;
```

Members

Members	Description
int16_t re;	real portion (a)
int16_t im;	imaginary portion (b)

Description

Q15 complex number (a + bi)

int32c Structure

File

[dsp.h](#)

C

```
typedef struct {
    int32_t re;
    int32_t im;
} int32c;
```

Members

Members	Description
int32_t re;	real portion (a)
int32_t im;	imaginary portion (b)

Description

Q31 complex number (a + bi)

matrix32 Structure

File

[dsp.h](#)

C

```
typedef struct {
    int32_t row;
    int32_t col;
    int32_t * pMatrix;
} matrix32;
```

Members

Members	Description
int32_t row;	matrix rows
int32_t col;	matrix columns
int32_t * pMatrix;	matrix pointer to data

Description

Q31 matrix

PARAM_EQUAL_FILTER Structure

File

[dsp.h](#)

C

```
typedef struct _PARAM_EQUAL_FILTER {
    PARAM_FILTER_GAIN G;
    int16_t log2Alpha;
    int16_t b[3];
    int16_t a[2];
    int32_t z[2];
} PARAM_EQUAL_FILTER;
```

Members

Members	Description
PARAM_FILTER_GAIN G;	Filter max gain multiplier

int16_t log2Alpha;	coefficient scaling bit shift value
int16_t b[3];	Feedforward Coefficients, Q15 format
int16_t a[2];	Feedback Coefficients, Q15 format
int32_t Z[2];	Filter memory, should be initialized to zero.

Description

IIR BQ filter structure Q15 data, Q31 storage

PARAM_EQUAL_FILTER_16 Structure

File

[dsp.h](#)

C

```
typedef struct _PARAM_EQUAL_FILTER_16 {
    PARAM_FILTER_GAIN G;
    int16_t log2Alpha;
    int16_t b[3];
    int16_t a[2];
    int16_t z[2];
} PARAM_EQUAL_FILTER_16;
```

Members

Members	Description
PARAM_FILTER_GAIN G;	Filter max gain multiplier
int16_t log2Alpha;	coefficient scaling bit shift value
int16_t b[3];	Feedforward Coefficients, Q15 format
int16_t a[2];	Feedback Coefficients, Q15 format
int16_t z[2];	Filter memory, should be initialized to zero.

Description

IIR BQ filter structure Q15

PARAM_EQUAL_FILTER_32 Structure

File

[dsp.h](#)

C

```
typedef struct _PARAM_EQUAL_FILTER_32 {
    PARAM_FILTER_GAIN G;
    int log2Alpha;
    int32_t b[3];
    int32_t a[2];
    int32_t z[2];
} PARAM_EQUAL_FILTER_32;
```

Members

Members	Description
PARAM_FILTER_GAIN G;	Filter max gain multiplier
int log2Alpha;	coefficient scaling bit shift value
int32_t b[3];	Feedforward Coefficients, Q15 format
int32_t a[2];	Feedback Coefficients, Q15 format
int32_t z[2];	Filter memory, should be initialized to zero.

Description

IIR BQ filter structure Q31

PARAM_FILTER_GAIN Structure

File

dsp.h

C

```
typedef struct {
    int16_t fracGain;
    int16_t expGain;
} PARAM_FILTER_GAIN;
```

Members

Members	Description
int16_t fracGain;	Q15 fractional filter gain
int16_t expGain;	log2N (binary) filter gain

Description

filter gain structure

MAX16 Macro

File

dsp.h

C

```
#define MAX16 ((int16_t) 0x7FFF) // maximum Q15
```

Description

maximum Q15

MAX32 Macro

File

dsp.h

C

```
#define MAX32 ((int32_t) 0x7FFFFFFF) // maximum Q31
```

Description

maximum Q31

MIN16 Macro

File

dsp.h

C

```
#define MIN16 ((int16_t) 0x8000) // minimum Q15
```

Description

minimum Q15

MIN32 Macro

File

dsp.h

C

```
#define MIN32 ((int32_t) 0x80000000) // minimum Q31
```

Description

minimum Q31

Files**Files**

Name	Description
dsp.h	DSP functions for the PIC32MZ device family.












Description

This section lists the source and header files used by the DSP Fixed-Point Math Library.


















dsp.h

DSP functions for the PIC32MZ device family.

Functions

	Name	Description
	DSP_ComplexAdd32	Calculates the sum of two complex numbers.
	DSP_ComplexConj16	Calculates the complex conjugate of a complex number.
	DSP_ComplexConj32	Calculates the complex conjugate of a complex number.
	DSP_ComplexDotProd32	Calculates the dot product of two complex numbers.
	DSP_ComplexMult32	Multiplies two complex numbers.
	DSP_ComplexScalarMult32	Multiplies a complex number and a scalar number.
	DSP_ComplexSub32	Calculates the difference of two complex numbers.
	DSP_FilterFIR32	Performs a Finite Infinite Response (FIR) filter on a vector.
	DSP_FilterFIRDecim32	Performs a decimating FIR filter on the input array.
	DSP_FilterFIRInterp32	Performs an interpolating FIR filter on the input array.
	DSP_FilterIIR16	Performs a single-sample cascaded biquad Infinite Impulse Response (IIR) filter.
	DSP_FilterIIRBQ16	Performs a single-pass IIR Biquad Filter.
	DSP_FilterIIRBQ16_cascade8	Performs a single-sample IIR Biquad Filter as a cascade of 8 series filters.
	DSP_FilterIIRBQ16_cascade8_fast	Performs a single-sample IIR Biquad Filter as a cascade of 8 series filters.
	DSP_FilterIIRBQ16_fast	Performs a single-pass IIR Biquad Filter.
	DSP_FilterIIRBQ16_parallel8	Performs a 8 parallel single-pass IIR Biquad Filters, and sums the result.
	DSP_FilterIIRBQ16_parallel8_fast	Performs a 8 parallel single-pass IIR Biquad Filters, and sums the result.
	DSP_FilterIIRBQ32	Performs a high resolution single-pass IIR Biquad Filter.
	DSP_FilterIIRSetup16	Converts biquad structure to coeffs array to set up IIR filter.
	DSP_FilterLMS16	Performs a single sample Least Mean Squares FIR Filter.
	DSP_MatrixAdd32	Addition of two matrices $C = (A + B)$.
	DSP_MatrixEqual32	Equality of two matrices $C = (A)$.
	DSP_MatrixInit32	Initializes the first N elements of a Matrix to the value num.
	DSP_MatrixMul32	Multiplication of two matrices $C = A \times B$.
	DSP_MatrixScale32	Scales each element of an input buffer (matrix) by a fixed number.
	DSP_MatrixSub32	Subtraction of two matrices $C = (A - B)$.
	DSP_MatrixTranspose32	Transpose of a Matrix $C = A (T)$.
	DSP_TransformFFT16	Creates an Fast Fourier Transform (FFT) from a time domain input.
	DSP_TransformFFT16_setup	Creates FFT coefficients for use in the FFT16 function.
	DSP_TransformFFT32	Creates an Fast Fourier Transform (FFT) from a time domain input.
	DSP_TransformFFT32_setup	Creates FFT coefficients for use in the FFT32 function.
	DSP_TransformIFFT16	Creates an Inverse Fast Fourier Transform (FFT) from a frequency domain input.
	DSP_TransformWindow_Bart16	Perform a Bartlett window on a vector.
	DSP_TransformWindow_Bart32	Perform a Bartlett window on a vector.




	DSP_TransformWindow_Black16	Perform a Blackman window on a vector.
	DSP_TransformWindow_Black32	Perform a Blackman window on a vector.
	DSP_TransformWindow_Cosine16	Perform a Cosine (Sine) window on a vector.
	DSP_TransformWindow_Cosine32	Perform a Cosine (Sine) window on a vector.
	DSP_TransformWindow_Hamm16	Perform a Hamming window on a vector.
	DSP_TransformWindow_Hamm32	Perform a Hamming window on a vector.
	DSP_TransformWindow_Hann16	Perform a Hanning window on a vector.
	DSP_TransformWindow_Hann32	Perform a Hanning window on a vector.
	DSP_TransformWindow_Kaiser16	Perform a Kaiser window on a vector.
	DSP_TransformWindow_Kaiser32	Perform a Kaiser window on a vector.
	DSP_TransformWinInit_Bart16	Create a Bartlett window.
	DSP_TransformWinInit_Bart32	Create a Bartlett window.
	DSP_TransformWinInit_Black16	Create a Blackman window.
	DSP_TransformWinInit_Black32	Create a Blackman window.
	DSP_TransformWinInit_Cosine16	Create a Cosine (Sine) window.
	DSP_TransformWinInit_Cosine32	Create a Cosine (Sine) window.
	DSP_TransformWinInit_Hamm16	Create a Hamming window.
	DSP_TransformWinInit_Hamm32	Create a Hamming window.
	DSP_TransformWinInit_Hann16	Create a Hanning window.
	DSP_TransformWinInit_Hann32	Create a Hanning window.
	DSP_TransformWinInit_Kaiser16	Create a Kaiser window.
	DSP_TransformWinInit_Kaiser32	Create a Kaiser window.
	DSP_VectorAbs16	Calculate the absolute value of a vector.
	DSP_VectorAbs32	Calculate the absolute value of a vector.
	DSP_VectorAdd16	Calculate the sum of two vectors.
	DSP_VectorAdd32	Calculate the sum of two vectors.
	DSP_VectorAddc16	Calculate the sum of a vector and a constant.
	DSP_VectorAddc32	Calculate the sum of a vector and a constant.
	DSP_VectorAutocorr16	Computes the Autocorrelation of a Vector.
	DSP_VectorBexp16	Computes the maximum binary exponent of a vector.
	DSP_VectorBexp32	Computes the maximum binary exponent of a vector.
	DSP_VectorChkEqu32	Compares two input vectors, returns an integer '1' if equal, and '0' if not equal.
	DSP_VectorCopy	Copies the elements of one vector to another.
	DSP_VectorCopyReverse32	Reverses the order of elements in one vector and copies them into another.
	DSP_VectorDivC	Divides the first N elements of inVector by a constant divisor, and stores the result in outVector.
	DSP_VectorDotp16	Computes the dot product of two vectors, and scales the output by a binary factor.
	DSP_VectorDotp32	Computes the dot product of two vectors, and scales the output by a binary factor
	DSP_VectorExp	Computes the EXP (e^x) of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorFill	Fills an input vector with scalar data.
	DSP_VectorLn	Computes the Natural Log, $\ln(x)$, of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorLog10	Computes the $\log_{10}(x)$, of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorLog2	Computes the $\log_2(x)$ of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorMax32	Returns the maximum value of a vector.
	DSP_VectorMaxIndex32	Returns the index of the maximum value of a vector.
	DSP_VectorMean32	Calculates the mean average of an input vector.
	DSP_VectorMin32	Returns the minimum value of a vector.
	DSP_VectorMinIndex32	Returns the index of the minimum value of a vector.
	DSP_VectorMul16	Multiplication of a series of numbers in one vector to another vector.
	DSP_VectorMul32	Multiplication of a series of numbers in one vector to another vector.
	DSP_VectorMulc16	Multiplication of a series of numbers in one vector to a scalar value.
	DSP_VectorMulc32	Multiplication of a series of numbers in one vector to a scalar value.
	DSP_VectorNegate	Inverses the sign (negates) the elements of a vector.
	DSP_VectorRecip	Computes the reciprocal ($1/x$) of the first N elements of inVector, and stores the result in outVector.

	DSP_VectorRMS16	Computes the root mean square (RMS) value of a vector.
	DSP_VectorShift	Shifts the data index of an input data vector.
	DSP_VectorSqrt	Computes the square root of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorStdDev16	Computes the Standard Deviation of a Vector.
	DSP_VectorSub16	Calculate the difference of two vectors.
	DSP_VectorSub32	Calculate the difference of two vectors.
	DSP_VectorSumSquares16	Computes the sum of squares of a vector, and scales the output by a binary factor.
	DSP_VectorSumSquares32	Computes the sum of squares of a vector, and scales the output by a binary factor.
	DSP_VectorVari16	Computes the variance of N elements of a Vector.
	DSP_VectorVariance	Computes the variance of N elements of inVector.
	DSP_VectorZeroPad	Fills an input vector with zeros.
	mul16	multiply and shift integer
	mul16r	multiply and shift Q15
	mul32	multiply and shift Q31
	SAT16	saturate both positive and negative Q15
	SAT16N	saturate negative Q15
	SAT16P	saturate positive Q15

Macros

	Name	Description
	MAX16	maximum Q15
	MAX32	maximum Q31
	MIN16	minimum Q15
	MIN32	minimum Q31

Structures

	Name	Description
	_PARAM_EQUAL_FILTER	IIR BQ filter structure Q15 data, Q31 storage
	_PARAM_EQUAL_FILTER_16	IIR BQ filter structure Q15
	_PARAM_EQUAL_FILTER_32	IIR BQ filter structure Q31
	biquad16	Q15 biquad
	int16c	Q15 complex number (a + bi)
	int32c	Q31 complex number (a + bi)
	matrix32	Q31 matrix
	PARAM_EQUAL_FILTER	IIR BQ filter structure Q15 data, Q31 storage
	PARAM_EQUAL_FILTER_16	IIR BQ filter structure Q15
	PARAM_EQUAL_FILTER_32	IIR BQ filter structure Q31
	PARAM_FILTER_GAIN	filter gain structure

Description

Digital Signal Processing (DSP) Library

The DSP Library provides functions that are optimized for performance on the PIC32MZ families of devices that have microAptiv core features and utilize DSP ASE. The library provides advanced mathematical operations for complex numbers, vector and matrix mathematics, digital filtering and transforms.

All functions are implemented in efficient assembly with C-callable prototypes. In some cases both 16-bit and 32-bit functions are supplied to enable the user with a choice of resolution and performance.

For most functions, input and output data is represented by 16-bit fractional numbers in Q15 format, which is the most commonly used data format for signal processing. Some functions use other data formats internally for increased precision of intermediate results.

The Q15 data type used by the DSP functions is specified as `int16_t` in the C header file that is supplied with the library. Note that within C code, care must be taken to avoid confusing fixed-point values with integers. To the C compiler, objects declared with `int16_t` type are integers, not fixed-point, and all arithmetic operations performed on those objects in C will be done as integers. Fixed-point values have been declared as `int16_t` only because the standard C language does not include intrinsic support for fixed-point data types.

Some functions also have versions operating on 32-bit fractional data in Q31 format. These functions operate similarly to their 16-bit counterparts. However, it should be noted that the 32-bit functions do not benefit much from the SIMD capabilities offered by DSP ASE. Thus, the performance of the 32-bit functions is generally reduced compared to the performance of the corresponding 16-bit functions.

Signed fixed point types are defined as follows:

Qn.m where:

- n is the number of data bits to the left of the radix point
- m is the number of data bits to the right of the radix point
- a signed bit is implied

Unique variable types for fractional representation are also defined:

Exact Name # Bits Required Type Q0.15 (Q15) 16 int16_t Q0.31 (Q31) 32 int32_t

Table of DSP Library functions:

Complex Math:

```
void DSP_ComplexAdd32(int32c *indata1, int32c *indata2, int32c *Output);
void DSP_ComplexConj16(int16c *indata, int16c *Output);
void DSP_ComplexConj32(int32c *indata, int32c *Output);
void DSP_ComplexDotProd32(int32c *indata1, int32c *indata2, int32c *Output);
void DSP_ComplexMult32(int32c *indata1, int32c *indata2, int32c *Output);
void DSP_ComplexScalarMult32(int32c *indata, int32_t Scalar, int32c *Output);
void DSP_ComplexSub32(int32c *indata1, int32c *indata2, int32c *Output);
```

Filter Functions:

```
void DSP_FilterFIR32(int32_t *outdata, int32_t *indata, int32_t *coeffs2x, int32_t *delayline, int N, int K, int scale);
void DSP_FilterFIRDecim32(int32_t *outdata, int32_t *indata, int32_t *coeffs, int32_t *delayline, int N, int K, int scale, int rate);
int16_t DSP_FilterIIR16(int16_t in, int16_t *coeffs, int16_t *delayline, int B, int scale);
void DSP_FilterIIRSetup16(int16_t *coeffs, biquad16 *bq, int B);
void DSP_FilterFIRInterp32(int32_t *outdata, int32_t *indata, int32_t *coeffs, int32_t *delayline, int N, int K, int scale, int rate);
int16_t DSP_FilterLMS16(int16_t in, int16_t ref, int16_t *coeffs, int16_t *delayline, int16_t *error, int K, int16_t mu);
int16_t DSP_FilterIIRBQ16_fast(int16_t Xin, PARM_EQUAL_FILTER_16 *pFilter);
int16_t DSP_FilterIIRBQ16(int16_t Xin, PARM_EQUAL_FILTER *pFilter);
int32_t DSP_FilterIIRBQ32(int32_t Xin, PARM_EQUAL_FILTER_32 *pFilter);
int16_t DSP_FilterIIRBQ16_cascade8(int16_t Xin, PARM_EQUAL_FILTER *pFilter_Array);
int16_t DSP_FilterIIRBQ16_cascade8_fast(int16_t Xin, PARM_EQUAL_FILTER_16 *pFilter_Array);
int16_t DSP_FilterIIRBQ16_parallel8(int16_t Xin, PARM_EQUAL_FILTER *pFilter);
int16_t DSP_FilterIIRBQ16_parallel8_fast(int16_t Xin, PARM_EQUAL_FILTER_16 *pFilter);
```

Matrix Functions:

```
void DSP_MatrixAdd32(matrix32 *resMat, matrix32 *srcMat1, matrix32 *srcMat2);
void DSP_MatrixEqual32(matrix32 *resMat, matrix32 *srcMat);
void DSP_MatrixInit32(int32_t *data_buffer, int32_t N, int32_t num);
void DSP_MatrixMul32(matrix32 *resMat, matrix32 *srcMat1, matrix32 *srcMat2);
void DSP_MatrixScale32(int32_t *data_buffer, int32_t N, int32_t num);
void DSP_MatrixSub32(matrix32 *resMat, matrix32 *srcMat1, matrix32 *srcMat2);
void DSP_MatrixTranspose32(matrix32 *desMat, matrix32 *srcMat);
```

Transforms:

```
void DSP_TransformFFT16(int16c *dout, int16c *din, int16c *twiddles, int16c *scratch, int log2N);
void DSP_TransformIFFT16(int16c *dout, int16c *din, int16c *twiddles, int16c *scratch, int log2N);
void DSP_TransformFFT16_setup(int16c *twiddles, int log2N);
void DSP_TransformFFT32(int32c *dout, int32c *din, int32c *twiddles, int32c *scratch, int log2N);
void DSP_TransformFFT32_setup(int32c *twiddles, int log2N);
void DSP_TransformWindow_Bart16(int16_t *OutVector, int16_t *InVector, int N);
void DSP_TransformWindow_Bart32(int32_t *OutVector, int32_t *InVector, int N);
void DSP_TransformWindow_Black16(int16_t *OutVector, int16_t *InVector, int N);
void DSP_TransformWindow_Black32(int32_t *OutVector, int32_t *InVector, int N);
void DSP_TransformWindow_Cosine16(int16_t *OutVector, int16_t *InVector, int N);
void DSP_TransformWindow_Cosine32(int32_t *OutVector, int32_t *InVector, int N);
void DSP_TransformWindow_Hamm16(int16_t *OutVector, int16_t *InVector, int N);
void DSP_TransformWindow_Hamm32(int32_t *OutVector, int32_t *InVector, int N);
void DSP_TransformWindow_Hann16(int16_t *OutVector, int16_t *InVector, int N);
void DSP_TransformWindow_Hann32(int32_t *OutVector, int32_t *InVector, int N);
void DSP_TransformWindow_Kaiser16(int16_t *OutVector, int16_t *InVector, int N);
```

```

void DSP_TransformWindow_Kaiser32(int32_t *OutVector, int32_t *InVector, int N);
void DSP_TransformWinInit_Bart16(int16_t *OutWindow, int N);
void DSP_TransformWinInit_Bart32(int32_t *OutWindow, int N);
void DSP_TransformWinInit_Black16(int16_t *OutWindow, int N);
void DSP_TransformWinInit_Black32(int32_t *OutWindow, int N);
void DSP_TransformWinInit_Cosine16(int16_t *OutWindow, int N);
void DSP_TransformWinInit_Cosine32(int32_t *OutWindow, int N);
void DSP_TransformWinInit_Hamm16(int16_t *OutWindow, int N);
void DSP_TransformWinInit_Hamm32(int32_t *OutWindow, int N);
void DSP_TransformWinInit_Hann16(int16_t *OutWindow, int N);
void DSP_TransformWinInit_Hann32(int32_t *OutWindow, int N);
void DSP_TransformWinInit_Kaiser16(int16_t *OutWindow, int N);
void DSP_TransformWinInit_Kaiser32(int32_t *OutWindow, int N);
Vector Math:
void DSP_VectorAbs16(int16_t *outdata, int16_t *indata, int N);
void DSP_VectorAbs32(int32_t *outdata, int32_t *indata, int N);
void DSP_VectorAdd16(int16_t *outdata, int16_t *indata1, int16_t *indata2, int N);
void DSP_VectorAdd32(int32_t *outdata, int32_t *indata1, int32_t *indata2, int N);
void DSP_VectorAddc16(int16_t *outdata, int16_t *indata, int16_t c, int N);
void DSP_VectorAddc32(int32_t *outdata, int32_t *indata, int32_t c, int N);
void DSP_VectorAutocorr16(int16_t *outCorr, int16_t *inVector, int N, int K);
int DSP_VectorBexp16(int16_t *DataIn, int N);
int DSP_VectorBexp32(int32_t *DataIn, int N);
int DSP_VectorChkEqu32(int32_t *indata1, int32_t *indata2, int N);
void DSP_VectorCopy(int32_t *outdata, int32_t *indata, int N);
void DSP_VectorCopyReverse32(int32_t *outdata, int32_t *indata, int N);
void DSP_VectorDivC(_Q16 *outVector, _Q16 *inVector, _Q16 divisor, int N);
int16_t DSP_VectorDotp16(int16_t *indata1, int16_t *indata2, int N, int scale);
int32_t DSP_VectorDotp32(int32_t *indata1, int32_t *indata2, int N, int scale);
void DSP_VectorExp(_Q16 *outVector, _Q16 *inVector, int N);
void DSP_VectorFill(int32_t *indata, int32_t data, int N);
void DSP_VectorLog10(_Q3_12 *outVector, _Q16 *inVector, int N);
void DSP_VectorLog2(_Q5_10 *outVector, _Q16 *inVector, int N);
void DSP_VectorLn(_Q4_11 *outVector, _Q16 *inVector, int N);
int32_t DSP_VectorMax32(int32_t *indata, int N);
int DSP_VectorMaxIndex32(int32_t *indata, int N);
int32_t DSP_VectorMean32(int32_t *indata, int N);
int32_t DSP_VectorMin32(int32_t *input, int N);
int DSP_VectorMinIndex32(int32_t *indata, int N);
void DSP_VectorMul16(int16_t *outdata, int16_t *indata1, int16_t *indata2, int N);
void DSP_VectorMul32(int32_t *outdata, int32_t *indata1, int32_t *indata2, int N);
void DSP_VectorMulc16(int16_t *outdata, int16_t *indata, int16_t c, int N);
void DSP_VectorMulc32(int32_t *outdata, int32_t *indata, int32_t c, int N);
void DSP_VectorNegate(int32_t *outdata, int32_t *indata, int N);
void DSP_VectorRecip(_Q16 *outVector, _Q16 *inVector, int N);
int16_t DSP_VectorRMS16(int16_t *inVector, int N);
void DSP_VectorShift(int32_t *outdata, int32_t *indata, int N, int shift);
int16_t DSP_VectorStdDev16(int16_t *inVector, int N);
void DSP_VectorSqrt(_Q16 *outVector, _Q16 *inVector, int N);
void DSP_VectorSub16(int16_t *outdata, int16_t *indata1, int16_t *indata2, int N);
void DSP_VectorSub32(int32_t *outdata, int32_t *indata1, int32_t *indata2, int N);
int16_t DSP_VectorSumSquares16(int16_t *indata, int N, int scale);
int32_t DSP_VectorSumSquares32(int32_t *indata, int N, int scale);
int16_t DSP_VectorVari16(int16_t *inVector, int N);

```



```
int32_t DSP_VectorVariance(int32_t *inVector, int N);
void DSP_VectorZeroPad(int32_t *indata, int N);
```

File Name

dsp.h

Company

Microchip Technology Inc.

LibQ Fixed-Point 'C' Math Library

This topic describes the LibQ Fixed-Point 'C' Math Library.

Introduction

The LibQ Fixed-Point 'C' Math Library is available for the PIC32 family of microcontrollers.

Description

The LibQ Fixed-Point 'C' Math Library provides fixed-point math functions written in C for portability between core processors.

The library utilizes signed fixed point types (fractional Q types specified by Q_n.m) as follows:

Q_nm where:

- n is the number of data bits to the left of the radix point
- m is the number of data bits to the right of the radix point
- a signed bit is implied (unless stated otherwise)

Whereas, the implied fractional scaling of the integer value is given by 2^{-(m)}, i.e. arithmetic left-shift of m bit positions of the fractional part into the value. This necessarily reduces the maximum magnitude of that can be stored in the n+m bit value, since only n bit remain and 1 of those bits is usually the sign..

For convenience, short names are also defined for arbitrary scaled fractional types:

- **q15** is signed fractional 16 bit value
- **q31** is signed fractional 32 bit value

In addition, A pseudo floating point 32 bit format (**FxQFloat32**) is defined that consists of 16 bit mantissa and a 16 bit exponent (base 2).

Functions in the library are prefixed with the type of the return value and followed by the argument types, as follows:

libq_: libq_q15_sin_Q2d13

For example, **libq_q1d15_Sin_q10d6** returns a Q1.15 value equal to the sine of an angle specified as a Q10.6 value (in degrees between 0 and 360). Argument types separated by an underscore.

For arbitrary scaled types (**q15** and **q31**), the scaling of the result will depend on the function and the scaling of the arguments. For instance, **libq_q15_Add_q15_q15(a,b)** will return a scaled value type that is the same as the two input types (which also must have equivalent Q format).

The library supports the original ETSI family of basic operations as described in:

"ETSI G.729 recommendation Coding of speech at 8 kbit/s using conjugate-structure algebraic-code excited linear prediction (CS-ACELP)".

This includes:

- Saturated arithmetic operations, such as L_add, L_sub, sub, add.
- Multiplication operations, such as mult, mult_r and L_mult.
- Arithmetic shift operations, such as shl and shr.
- Data conversion operations, such as extract_l, extract_h, and round.
- Normalization to maximize resolution, such as norm_l
- Loading of 15 bit values into 32 bit values, such as L_deposit_h
- Signed negation, such as L_negate
- Multiply accumulate, such as L_mac, L_macNs, mac_r
- Multiply subtract, such as L_msu, L_msuNs msu_r
- Divide, such as div_s
- Saturate value, such as L_sat
- Rounding, such as round.
- Left Shift, such as L_shl and shl
- Right Shift, such as L_shr, L_shr_r and shr
- Extraction of 16 bit values, such as extract_l
- Absolute value, such as L_abs

The 'C' Code implementation provides portability of the library, yet the functions easily translate to specific processor assembly code (intrinsic operations) when needed.

**Note:**

The LibQ Fixed-Point 'C' Math Library functions do not correspond to the LibQ Fixed-Point Math Library, which is optimized for the microAptiv core processor and is written in Assembly language.

Using the Library

This topic describes the basic architecture of the LibQ Fixed-Point 'C' Math Library and provides information and examples on its use.

Description

Interface Header File: [libq_c.h](#)

The interface to the LibQ Fixed-Point 'C' Math Library is defined in the [libq_c.h](#) header file. Any C language source (.c) file that uses the LibQ Fixed-Point Library should include [libq_c.h](#).

Library File:

The LibQ Fixed-Point 'C' Math Library archive (.a) file is installed with MPLAB Harmony.

Library Interface

Functions

	Name	Description
⇒	libq_q15_Abs_q15	Saturated Absolute value.
⇒	libq_q15_Add_q15_q15	Add two 16-bit 2s-complement fractional values.
⇒	libq_q15_DivisionWithSaturation_q15_q15	Fractional division with saturation.
⇒	libq_q15_ExtractH_q31	Extracts upper 16 bits of input 32-bit fractional value.
⇒	libq_q15_ExtractL_q31	Extracts lower 16-bits of input 32-bit fractional value. Descriptionf Extracts lower 16-bits of input 32-bit fractional value and returns them as 16-bit fractional value. This is a bit-for-bit extraction of the bottom 16-bits of the 32-bit input. This function relates to the ETSI extract_l function.
⇒	libq_q15_Negate_q15	Negate 16-bit 2s-complement fractional value with saturation.
⇒	libq_q15_MacR_q31_q15_q15	Multiply accumulate with rounding.
⇒	libq_q15_RoundL_q31	Rounds the lower 16-bits of the 32-bit fractional input.
⇒	libq_q15_MsuR_q31_q15_q15	Multiply-Subtraction with rounding
⇒	libq_q1d15_Sin_q10d6	Approximates the sine of an angle.
⇒	libq_q31_Abs_q31	Saturated Absolute value.
⇒	libq_q31_Add_q31_q31	Add two 32-bit 2s-complement fractional values.
⇒	libq_q31_DepositH_q15	Place 16 bits in the upper half of 32 bit word.
⇒	libq_q15_Sub_q15_q15	Subtract two 16-bit 2s-complement fractional values
⇒	libq_q31_DepositL_q15	Place 16 bits in the lower half of 32 bit word.
⇒	libq_q31_Multi_q15_q31	Implement 16 bit by 32 bit multiply.
⇒	libq_q31_Negate_q31	Negate 32-bit 2s-complement fractional value with saturation.
⇒	libq_q15_ExpAvg_q15_q15_q1d15	Exponential averaging
⇒	libq_q31_Mac_q31_q15_q15	Multiply-Accumulate function WITH saturation
⇒	libq_q31_Msu_q31_q15_q15	L_msu(a,b,c)
⇒	libq_q31_ShiftLeft_q31_i16	'Arithmetic' Shift of the 32-bit value.
⇒	libq_q20d12_Sin_q20d12	3rd order Polynomial appr. of a sine function
⇒	Fx16Norm	Normalize the 16-bit fractional value.
⇒	Fx32Norm	Normalize the 32-bit number.
⇒	libq_q15_ShiftLeft_q15_i16	'Arithmetic' Shift of the 16-bit input argument.
⇒	libq_q15_ShiftRight_q15_i16	'Arithmetic' RIGHT Shift on a 16-bit value.
⇒	libq_q15_ShiftRightRound_q15_i16	Performs an 'Arithmetic' RIGHT Shift on a 16-bit input.
⇒	libq_q31_Mult2_q15_q15	fractional multiplication of two 16-bit fractional values.
⇒	libq_q31_ShiftRight_q31_i16	'Arithmetic' RIGHT Shift on a 32-bit value.
⇒	libq_q31_ShiftRightRound_q31_i16	'Arithmetic' RIGHT Shift on a 32-bit value
⇒	libq_q31_Sub_q31_q31	Subtract two 32-bit 2s-complement fractional values

Data Types and Constants

Name	Description
q15	q15 n.n (signed)
q31	q31 n.n (signed)
q63	Q63 n.n (signed)
<code>_LIBQ_C_H_</code>	This is macro <code>_LIBQ_C_H_</code> .
BITMASKFRACT16	Bit Mask for 16
BITMASKFRACT32	Bit Mask for 32
FI2Fract16	Converts floating point constant value to fractional 16-bit value
FI2Fract32	Converts floating point constant value to fractional 32-bit value
FI2FxFnt	Converts floating point constant value to fixed/fractional value
FI2FxFnt16	Converts floating point constant value to fixed point 16-bit value
FI2FxFnt32	Converts floating point constant value to fixed point 32-bit value
FI2Int16	Converts floating point constant expression to 16-bit integer value
FI2Int32	Converts floating point constant expression to 32-bit integer value
FrMax	find the maximum of two numbers
FrMin	find the minimum of two numbers
LOG102Q5D11	log ₁₀ (2) scaled to Q5.11 format
MAXFRACT16	0.999969
MAXFRACT32	0.9999999995
MAXINT16	Maximum and minimum values for 16-bit data types.
MAXINT32	This is macro MAXINT32.
MAXPFLOAT32	minimum and maximum definitions for FX floating point data types
MINFRACT16	1.000000
MINFRACT32	1.0000000000
MININT16	This is macro MININT16.
MININT32	This is macro MININT32.
MINPFLOAT32	This is macro MINPFLOAT32.
MSBBITFRACT16	16-bit Sign Bit
MSBBITFRACT32	32-bit Sign Bit
NINETYQ10D22	Ninety degrees scaled to Q10d22
NINETYQ10D6	Ninety degrees scaled to Q10d6
NORMNEGFRACT16	Max -val for 16
NORMNEGFRACT32	Max -val for 32
NORMPOSFRACT16	Min +val for 16
NORMPOSFRACT32	Min +val for 32
NUMBITSFRACT16	Num of bits 16
NUMBITSFRACT32	Num of bits 32
ONEEIGHTYQ10D22	180 degrees scaled to Q10d22
ONEEIGHTYQ10D6	180 degrees scaled to Q10d6
ROUNDFRACT32	Rounding value
THREESIXTYQ10D22	360 degrees scaled to Q10d22
THREESIXTYQ10D6	360 degrees scaled to Q10d6
TWOSEVENTYQ10D22	270 degrees scaled to Q10d22
TWOSEVENTYQ10D6	270 degrees scaled to Q10d6
UNITYFLOAT	This is macro UNITYFLOAT.
FxQFloat32	FxQFloat32 pseudo floating point type (limited floating point)
Exponent16ToQFloat32	Converts a power of 2 16-bit integer to 32-bit floating point value.
FI2QFloat32	Converts a decimal floating-point constant expression to pseudo float
i16	Q16d0

Description

This section describes the Application Programming Interface (API) functions, macros, and types of the LibQ Fixed Point 'C' Math Library. Refer to each section for a detailed description.

Functions

libq_q15_Abs_q15 Function

Saturated Absolute value.

File

[libq_C.h](#)

C

```
q15 libq_q15_Abs_q15(q15);
```

Returns

q15 result - abs(input) <= MAXFRACT16

Description

Function libq_q15_Abs_q15:

Creates a saturated Absolute value. It takes the absolute value of the 16-bit 2s-complement fractional input with saturation. The saturation is for handling the case where taking the absolute value of MINFRACT16 is greater than MAXFRACT16, or the allowable range of 16-bit values. This function relates to the ETSI abs function.

Parameters

Parameters	Description
q15 a	input argument

libq_q15_Add_q15_q15 Function

Add two 16-bit 2s-complement fractional values.

File

[libq_C.h](#)

C

```
q15 libq_q15_Add_q15_q15(q15, q15);
```

Returns

q15 - a+b on Range: MINFRACT16 <= result <= MAXFRACT16

Description

Function libq_q15_Add_q15_q15: f

Add two 16-bit 2s-complement fractional (op1 + op2) to produce a 16-bit 2s-complement fractional result with saturation. The saturation is for handling the overflow/underflow cases, where the result is set to MAX16 when an overflow occurs and the result is set to MIN16 when an underflow occurs. This function does not produce any status flag to indicate when an overflow or underflow has occurred. It is assumed that the binary point is in exactly the same bit position for both 16-bit inputs and the resulting 16-bit output.

libq_q15_DivisionWithSaturation_q15_q15 Function

Fractional division with saturation.

File

[libq_C.h](#)

C

```
q15 libq_q15_DivisionWithSaturation_q15_q15(q15, q15);
```

Returns

q15 result - ratio a/b in 16-bit fractional format

Description

Function `libq_q15_DivisionWithSaturation_q15_q15()`:

Performs fractional division with saturation. There are three restrictions that the calling code must satisfy.

1. Both the numerator and denominator must be positive.
2. In order to obtain a non-saturated result, the numerator must be LESS than or equal to the denominator.
3. The denominator must not equal zero.

If 'num' equals 'den', then the result equals [MAXINT16](#).

This function relates to the ETSI `div_s` function.

Parameters

Parameters	Description
q15 num	16-bit fractional numerator
q15 den	16-bit fractional denominator

libq_q15_ExtractH_q31 Function

Extracts upper 16 bits of input 32-bit fractional value.

File

[libq_C.h](#)

C

```
q15 libq_q15_ExtractH_q31(q31);
```

Returns

q15 result - Upper 16 bits of 32-bit argument a

Description

Function `libq_q15_ExtractH_q31`:

Extracts upper 16 bits of input 32-bit fractional value and returns them as 16-bit fractional value. This is a bit-for-bit extraction of the top 16-bits of the 32-bit input. This function relates to the ETSI `extract_h` function.

libq_q15_ExtractL_q31 Function

Extracts lower 16-bits of input 32-bit fractional value.

Descriptionf Extracts lower 16-bits of input 32-bit fractional value and returns them as 16-bit fractional value. This is a bit-for-bit extraction of the bottom 16-bits of the 32-bit input. This function relates to the ETSI `extract_l` function.

File

[libq_C.h](#)

C

```
q15 libq_q15_ExtractL_q31(q31);
```

Returns

q15 - Lower 16 bits of 32-bit argument a

Description

Function `libq_q15_ExtractL_q31`:

libq_q15_Negate_q15 Function

Negate 16-bit 2s-complement fractional value with saturation.

File

[libq_C.h](#)

C

```
q15 libq_q15_Negate_q15(q15);
```

Returns

q15 result on range: `MINFRACT16` <= result <= `MAXFRACT16`

Description

Function `libq_q15_Negate_q15`:

Negate 16-bit 2s-complement fractional value with saturation. The saturation is for handling the case where negating a `MINFRACT16` is greater than `MAXFRACT16`, or the allowable range of values. This function relates to the ETSI negate function.

libq_q15_MacR_q31_q15_q15 Function

Multiply accumulate with rounding.

File

[libq_C.h](#)

C

```
q15 libq_q15_MacR_q31_q15_q15(q31, q15, q15);
```

Returns

q15 result - a+b*c rounded

Description

Function `libq_q15_MacR_q31_q15_q15`:

This function is multiply-accumulate WITH Rounding applied to the accumulator result before it is saturated and the top 16-bits taken. This function first multiplies the two 16-bit input values 'b x c' which results in a 32-bit value. This result is left shifted by one to account for the extra sign bit inherent in the fractional-type multiply. So, the shifted number now has a '0' in the Lsb. The shifted multiplier output is then added to the 32-bit fractional input 'a'. Then the 32-bits of the accumulator output are rounded by adding '2^15'. This value is then saturated to be within the q15 range. It is assumed that the binary point of the 32-bit input value a is in the same bit position as the shifted multiplier output. This function is for fractional Qtype format data only and it therefore will not give the correct results for true integers.

This function relates to the ETSI `L_mac_r` function.

Parameters

Parameters	Description
q31 a	32-bit accumulator operand
q15 b	16-bit multiplication operand
q15 c	16-bit multiplication operand

libq_q15_RoundL_q31 Function

Rounds the lower 16-bits of the 32-bit fractional input.

File

[libq_C.h](#)

C

```
q15 libq_q15_RoundL_q31(q31);
```

Returns

q15 result

Description

Function `libq_q15_RoundL_q31`:

Rounds the lower 16-bits of the 32-bit fractional input into a 16-bit fractional value with saturation. This converts the 32-bit fractional value to 16-bit fractional value with rounding. This function calls the 'Add' function to perform the 32-bit rounding of the input value and 'ExtractH' function to extract to top 16-bits. This has the effect of rounding positive fractional values up and more positive, and has the effect of rounding negative fractional values up and more positive. This function relates to the ETSI round function.

libq_q15_MsuR_q31_q15_q15 Function

Multiply-Subtraction with rounding

File[libq_C.h](#)**C**

```
q15 libq_q15_MsuR_q31_q15_q15(q31, q15, q15);
```

Returns

q15 result - a-b*c rounded to Q1.15

Description

Function libq_q15_MsuR_q31_q15_q15:

This function is like Multiply-Subtract but WITH Rounding applied to the subtractor result before it is saturated and the top 16-bits taken. This function first multiplies the two 16-bit input values 'b x c' which results in a 32-bit value. This result is left shifted by one to account for the extra sign bit inherent in the fractional-type multiply. So, the shifted number now has a '0' in the Lsb. The shifted multiplier output is then SUBTRACTED From the 32-bit fractional input 'a'. Then the 32-bits output From this subtraction are rounded by adding '2^15'. This value is then saturated to be within the q15 range. It is assumed that the binary point of the 32-bit input value a is in the same bit position as the shifted multiplier output.

This function is for fractional Q-type format data only and it therefore will not give the correct results for true integers.

This function relates to the ETSI msu_r function.

Parameters

Parameters	Description
q31 a	Value which is subtracted from
q15 b	multiplication operand 1
q15 c	multiplication operand 2

libq_q1d15_Sin_q10d6 Function

Approximates the sine of an angle.

File[libq_C.h](#)**C**

```
q15 libq_q1d15_Sin_q10d6(q15 angleQ10d6);
```

Returns

q15 sine(angle) value in Q1.15

Description

Function libq_q1d15_Sin_q10d6:

This function approximates the sine of an angle using the following algorithm: $\sin(x) = 3.140625x + 0.02026367x^2 - 5.325196x^3 + 0.5446778x^4 + 1.800293x^5$. The approximation is accurate for any value of x from 0 degrees to 90 degrees. Because $\sin(-x) = -\sin(x)$ and $\sin(x) = \sin(180 - x)$, the sine of any angle can be inferred from an angle in the first quadrant. Therefore, any angle > 90 is converted to an angle between 0 & 90. The coefficients of the algorithm have been scaled by 1/8 to fit a Q1d15 format. So the result is scaled up by 8 to obtain the proper magnitudes. The algorithm expects the angle to be in degrees and represented in Q10.6 format. The computed sine value is returned in Q1.15 format.

Preconditions

none.

Parameters

Parameters	Description
q15 angle	The angle in degrees for which the sine is computed in Q10.6

libq_q31_Abs_q31 Function

Saturated Absolute value.

File[libq_C.h](#)

C

```
q31 libq_q31_Abs_q31(q31);
```

Returns

q31 result - abs(a) <= MAXFRACT32

Description

Function libq_q31_Abs_q31:

Creates a saturated Absolute value. It takes the absolute value of the 32-bit 2s-complement fractional input with saturation. The saturation is for handling the case where taking the absolute value of MINFRACT32 is greater than MAXFRACT32, or the allowable range of 32-bit values.

This function relates to the ETSI L_abs function.

libq_q31_Add_q31_q31 Function

Add two 32-bit 2s-complement fractional values.

File

libq_C.h

C

```
q31 libq_q31_Add_q31_q31(q31, q31);
```

Returns

q31 result a+b on range: MINFRACT32 <= result <= MAXFRACT32

Description

Function libq_q31_Add_q31_q31:

Add two 32-bit 2s-complement fractional (op1 + op2) to produce a 32-bit 2s-complement fractional result with saturation. The saturation is for handling the overflow/underflow cases, where the result is set to MAX32 when an overflow occurs and the result is set to MIN32 when an underflow occurs. This function does not produce any status flag to indicate when an overflow or underflow has occurred. It is assumed that the binary point is in exactly the same bit position for both 32-bit inputs and the resulting 32-bit output. This function relates to the ETSI L_add function.

libq_q31_DepositH_q15 Function

Place 16 bits in the upper half of 32 bit word.

File

libq_C.h

C

```
q31 libq_q31_DepositH_q15(q15);
```

Returns

q31 result 16-bits of a in upper MSB's and zeros in the lower LSB's

Description

Function libq_q31_DepositH_q15:

Composes a 32-bit fractional value by placing the input 16-bit fractional value in the composite MSB's and zeros the composite 16-bit LSB's This is a bit-for-bit placement of input 16-bits into the upper part of 32-bit result.

This function relates to the ETSI L_deposit_H function.

libq_q15_Sub_q15_q15 Function

Subtract two 16-bit 2s-complement fractional values

File

libq_C.h

C

```
q15 libq_q15_Sub_q15_q15(q15, q15);
```


Returns

q15 result a+b on range: `MINFRACT16` <= result <= `MAXFRACT16`

Description

Function `libq_q15_Sub_q15_q15`:

Subtract two 16-bit 2s-complement fractional (`op1 - op2`) to produce a 16-bit 2s-complement fractional difference result with saturation. The saturation is for handling the overflow/underflow cases, where the result is set to `MAX16` when an overflow occurs and the result is set to `MIN16` when an underflow occurs. This function does not produce any status flag to indicate when an overflow or underflow has occurred. It is assumed that the binary point is in exactly the same bit position for both 16-bit inputs and the resulting 16-bit output. This function relates to the ETSI sub function.

libq_q31_DepositL_q15 Function

Place 16 bits in the lower half of 32 bit word.

File

`libq_C.h`

C

```
q31 libq_q31_DepositL_q15(q15);
```

Returns

q31 result - SignExtended 16-bit MSB's and a Value in lower 16-bit LSB's

Description

Function `libq_q31_DepositL_q15`:

Composes a 32-bit fractional value by placing the 16-bit Fraction input value into the lower 16-bits of the 32-bit composite value. The 16-bit MSB's of the composite output are sign extended. This is a bit-for-bit placement of input 16-bits into the bottom portion of the composite 32-bit result with sign extension. This function relates to the ETSI `L_deposit_l` function.

libq_q31_Multi_q15_q31 Function

Implement 16 bit by 32 bit multiply.

File

`libq_C.h`

C

```
q31 libq_q31_Multi_q15_q31(q15 argAQ1d15, q31 argBQ1d31);
```

Returns

q31 result - a*b rounded

Description

Function `libq_q31_Multi_q15_q31()`:

Implement 16 bit by 32 bit multiply as shown below The 's' and 'u' notation shows the processing of signed and unsigned numbers.

-B1- -B0- s u 2nd argument is 32 bits -A0- s 1st argument is 16 bits

A0B0 A0B0 s=s*u 1st 32-bit product is A0*B0 A0B1 A0B1 s=s*s 2nd 32-bit product is A0*B1

-S2- -S1- -S0- s=s+s 48-bit result is the sum of products -P1- -P0- 32-bit return is the most significant bits of sum

The algorithm is implemented entirely with the fractional arithmetic library. The unsigned by signed multiply is implemented by shifting bits 15:1 to bits 14:0 of a 16-bit positive fractional number, which throws away bit 0 of the 32-bit number. Since that affects result bits that are used for rounding, rounding processing is included. Saturation processing is handled implicitly in the fractional arithmetic library, except for the case of maximum negative numbers.

libq_q31_Negate_q31 Function

Negate 32-bit 2s-complement fractional value with saturation.

File

[libq_C.h](#)

C

```
q31 libq_q31_Negate_q31(q31);
```

Returns

q31 result on range: [MINFRACT32](#) <= result <= [MAXFRACT32](#)

Description

Function libq_q31_Negate_q31:

Negate 32-bit 2s-complement fractional value with saturation. The saturation is for handling the case where negating a [MINFRACT32](#) is greater than [MAXFRACT32](#), or the allowable range of values. This function relates to the ETSI L_negate function.

libq_q15_ExpAvg_q15_q15_q1d15 Function

Exponential averaging

File

[libq_C.h](#)

C

```
q15 libq_q15_ExpAvg_q15_q15_q1d15(q15 prevAvgQ15, q15 newMeasQ15, q15 lamdaQ1d15);
```

Returns

q15 result - $S(k+1) = S(k)*L + X(k)*(1-L)$

Description

Function libq_q15_ExpAvg_q15_q15_q1d15()

Exponential averaging implements a smoothing function based on the form: $avg[i+1] = avg[i] * lamda + new * (1-lamda)$ In this implementation, it has been optimized as follows. $avg[i+1] = (avg[i] - new) * lamda + new$

The optimization precludes accurate processing of new numbers that differ from the current average by more than unity. If the difference is greater than unity or less than negative unity, the difference is saturated.

The effect is akin to a smaller lambda, e.g., the new value will have a greater weight than expected. If the smoothing is of data that is entirely positive or entirely negative, then the saturation will not be an issue.

Parameters

Parameters	Description
q15 S(k)	Previous exponential average
q15 X(k)	New value to be averaged in
q15 L	exponential averaging constant in Q1.15

libq_q31_Mac_q31_q15_q15 Function

Multiply-Accumulate function WITH saturation

File

[libq_C.h](#)

C

```
q31 libq_q31_Mac_q31_q15_q15(q31, q15, q15);
```

Returns

q31 result, a+b*c saturated

Description

Function `libq_q31_Mac_q31_q15_q15()`:

Performs a Multiply-Accumulate function WITH saturation. This routine returns the fully fractional 32-bit result From the accumulator output 'SAT(addOut_Q1d31)=outQ1d15' where 'multOut_Q1d31 + a_Q1d31 = addOut_Q1d31', and 'b_Q1d15 x c_Q1d15 = multOut_Q1d31'. The multiply is performed on the two 16-bit fully fully fractional input values 'b x c' which results in a 32-bit value. This result is left shifted by one to account for the extra sign bit inherent in the fully fully fractional-type multiply. The shifted number represents a Q1d31 number with the lsb set to '0'. This Q1d31 number is added with the 32-bit fully fully fractional input argument 'a'. Saturation is applied on the output of the accumulator to keep the value within the 32-bit fully fractional range and then this value is returned. This function is for fully fractional Q-type format data only and it therefore will not give the correct results for true integers.

This function relates to the ETSI `L_mac` function.

Parameters

Parameters	Description
q31 a	32-bit accumulator operand 1 in Q1d31
q15 b	16-bit multiplication operand 1 in Q1d15
q15 c	16-bit multiplication operand 2 in Q1d15

libq_q31_Msu_q31_q15_q15 Function

File

[libq_C.h](#)

C

```
q31 libq_q31_Msu_q31_q15_q15(q31, q15, q15);
```

Description

`L_msu(a,b,c)`

libq_q31_ShiftLeft_q31_i16 Function

'Arithmetic' Shift of the 32-bit value.

File

[libq_C.h](#)

C

```
q31 libq_q31_ShiftLeft_q31_i16(q31, i16);
```

Returns

q31 result - arithmetically shifted 32-bit signed integer output

Description

Function `libq_q31_ShiftLeft_q31_i16`:

Performs an 'Arithmetic' Shift of the 32-bit input argument 'a' left by the input argument 'b' bit positions. If 'b' is a positive number, a 32-bit left shift is performed with 'zeros' inserted to the right of the shifted bits. If 'b' is a negative number, a 32-bit right shift by b bit positions with 'sign extension':
 positive value: # of bits to left shift (zeros inserted at LSB's)
 negative value: # of bits to right shift (sign extend)

Saturation is applied if shifting causes an overflow or an underflow.

This function relates to the ETSI `L_shl` function.

Parameters

Parameters	Description
q31 a	32-bit signed integer value to be shifted
i16	16-bit signed integer shift index

libq_q20d12_Sin_q20d12 Function

3rd order Polynomial apprx. of a sine function

File[libq_C.h](#)**C**

```
q31 libq_q20d12_sin_q20d12(q31 angQ20d12);
```

Returns

q31 Sine(angle) value in Q20.12

Description

Function libq_q20d12_Sin_q20d12:
3rd order Polynomial appr. of a sine function

Preconditions

none.

Parameters

Parameters	Description
q31 angle	The angle in radians for which the sine is computed in Q20.12

Fx16Norm Function

Normalize the 16-bit fractional value.

File[libq_C.h](#)**C**

```
int16_t Fx16Norm(q15);
```

Returns

i16 result - The number of left shifts required to normalize

Description

Function Fx16Norm:

Produces then number of left shifts needed to Normalize the 16-bit fully fractional input. If the input 'a' is a positive number, it will produce the number of left shifts required to normalized it to the range of a minimum of $[(MAXFRAC16+1)/2]$ to a maximum of $[MAXFRAC16]$. If the input 'a' is a negative number, it will produce the number of left shifts required to normalized it to the range of a minimum of $[MINFRAC16]$ to a maximum of $[MINFRAC16/2]$. This function does not actually normalize the input, it just produces the number of left shifts required. To actually normalize the value the left shift function should be used with the value returned From this function.

the 16-bit input on range: $0 \Rightarrow \text{result} < 16$ (i.e. $NUMBITSFRAC16$) If $a > 0$: $0x4000 > \text{Normalized Value} \leq 0x7fff$ i.e. $(MAXFRAC16+1)/2 > aNorm \leq MAXFRAC16$ If $a < 0$: $0x8000 \geq \text{Normalized Value} < 0xC000$ i.e. $MINFRAC16 \geq aNorm < MINFRAC16/2$

This function relates to the ETSI norm_s function.

Parameters

Parameters	Description
q15	a in Q1.15

Fx32Norm Function

Normalize the 32-bit number.

File[libq_C.h](#)**C**

```
int16_t Fx32Norm(q31);
```

Returns

int16_t result - The number of left shifts required to normalize the

Description

Function Fx32Norm:

Produces then number of left shifts needed to Normalize the 32-bit fractional input. If the input 'a' is a positive number, it will produce the number of left shifts required to normalized it to the range of a minimum of $[(MAXFRACT32+1)/2]$ to a maximum of $[MAXFRACT32]$. If the input 'a' is a negative number, it will produce the number of left shifts required to normalized it to the range of a minimum of $[MINFRACT32]$ to a maximum of $[MINFRACT32/2]$. This function does not actually normalize the input, it just produces the number of left shifts required. To actually normalize the value the left-shift function should be used with the value returned From this function.

32-bit input on range: $0 \Rightarrow \text{result} < 32$ (i.e. $NUMBITSFRACT32$) If $a > 0$: $0x40000000 > \text{Normalized Value} \leq 0x7ffffff$ i.e. $(MAXFRACT32+1)/2 > aNorm \leq MAXFRACT32$ If $a < 0$: $0x80000000 \geq \text{Normalized Value} < 0xC0000000$ i.e. $MINFRACT32 \geq aNorm < MINFRACT32/2$

This function relates to the ETSI norm_l function.

Parameters

Parameters	Description
q31 a	32-bit Q1.d31 to be normalized

libq_q15_ShiftLeft_q15_i16 Function

'Arithmetic' Shift of the 16-bit input argument.

File

libq_C.h

C

```
q15 libq_q15_ShiftLeft_q15_i16(q15, i16);
```

Returns

q15 result - arithmetically shifted 16-bit signed integer output

Description

Function libq_q15_ShiftLeft_q15_i16:

Performs an 'Arithmetic' Shift of the 16-bit input argument 'a' left by the input argument 'b' bit positions. If 'b' is a positive number, a 16-bit left shift is performed with 'zeros' inserted to the right of the shifted bits. If 'b' is a negative number, a right shift by $\text{abs}(b)$ positions with 'sign extension' Saturation is applied if shifting causes an overflow or an underflow.

positive value: # of bits to left shift (zeros inserted at LSB's) (To not always saturate: if 'a=0', then $\text{max } b=15$, else $\text{max } b=14$) negative value: # of bits to right shift (sign extend)

This function relates to the ETSI shl function.

Parameters

Parameters	Description
q15 a	16-bit signed integer value to be shifted.
i16 b	16-bit signed integer shift value

libq_q15_ShiftRight_q15_i16 Function

'Arithmetic' RIGHT Shift on a 16-bit value.

File

libq_C.h

C

```
q15 libq_q15_ShiftRight_q15_i16(q15, i16);
```

Returns

q15 result - 16-bit signed shifted output

Description

Function libq_q15_ShiftRight_q15_i16:

Performs an 'Arithmetic' RIGHT Shift on a 16-bit input by 'b' bit positions. For positive shift directions ($b > 0$), 'b' Lsb-bits are shifted out to the right and 'b' sign-extended Msb-bits fill in From the left. For negative shift directions ($b < 0$), 'b' Lsb's are shifted to the LEFT with 0's filling in the empty lsb position. The left shifting causes 'b' Msb-bits to fall off to the left, saturation is applied to any shift left value that overflows. This function calls

the left-shift function to perform any 16-bit left shifts. This function does not provide any status-type information to indicate when overflows occur.
 positive value: # of bits to right shift (sign extend) { To get all sign bits, b>=15 } negative value: # of bits to left shift (zeros inserted at LSB's)
 This function relates to the ETSI shr function.

Parameters

Parameters	Description
q15 a	16-bit signed input value to shift
i16 b	16-bit signed integer shift index

libq_q15_ShiftRightRound_q15_i16 Function

Performs an 'Arithmetic' RIGHT Shift on a 16-bit input.

File

[libq_C.h](#)

C

```
q15 libq_q15_ShiftRightRound_q15_i16(q15, i16);
```

Returns

q15 result - Arithmetically shifted 16-bit signed integer output

Description

Function `libq_q15_ShiftRightRound_q15_q15`:

Performs an 'Arithmetic' RIGHT Shift on a 16-bit input by 'b' bits with Rounding applied. The rounding occurs by adding a bit weight of "1/2 Lsb", where the "Lsb" is the Ending (shifted) Lsb. For example: The initial Bit#(b) is after the right shift Bit#(0), so the rounding bit weight is Bit#(b-1). Rounding does not occur on either left shifts or on no shift needed cases. For positive shift directions (b>0), 'b' Lsb-bits are shifted out to the right and 'b' sign-extended Msb-bits fill in From the left. For negative shift directions (b<0), 'b' Lsb's are shifted to the LEFT with 0's filling in the empty lsb position. The left shifting causes 'b' Msb-bits to fall off to the left, saturation is applied to any shift left value that overflows. This function calls the left-shift function to perform the actual 16-bit left shift. This function does not provide any status-type information to indicate when overflows occur.

positive value: # of bits to right shift (sign extend) {b > 15, results in all sign bits} negative value: # of bits to left shift (zeros inserted at LSB's)
 This function relates to the ETSI shr_r function.

Parameters

Parameters	Description
q15 a	16-bit signed integer value to be shifted
i16 b	16-bit signed integer shift index

libq_q31_Mult2_q15_q15 Function

fractional multiplication of two 16-bit fractional values.

File

[libq_C.h](#)

C

```
q31 libq_q31_Mult2_q15_q15(q15, q15);
```

Returns

q31, a*b

Description

Function `libq_q31_Mult2_q15_q15`:

Performs fractional multiplication of two 16-bit fractional values and returns the 32-bit fractional scaled result. The function performs the Q15xQ15->Q30 fractional bit multiply. It then shifts the result left by '1', to give a Q31 type result, (the lsb is zero-filled). This automatic shift left is done to get rid of the extra sign bit that occurs in the interpretation of the fractional multiply result. Saturation is applied to any results that overflow, and then the function returns the 32-bit fractional q31 result. This function is for fractional 'Q' data only and it therefore will not give correct results for true integers (because left shift by '1'). For the special case where both inputs equal the [MINFRACT16](#), the function returns a value equal to MAXFACT32, i.e. 0x7fffffff = libq_q15_mult_q15_q31(0x8000,0x8000).

This function relates to the ETSI L_mult function.

Parameters

Parameters	Description
q15 a	multiplicand a
q15 b	multiplicand b

libq_q31_ShiftRight_q31_i16 Function

'Arithmetic' RIGHT Shift on a 32-bit value.

File

[libq_C.h](#)

C

```
q31 libq_q31_ShiftRight_q31_i16(q31, i16);
```

Returns

q31 result - Arithmetically shifted 32-bit signed integer output

Description

Function `libq_q31_ShiftRight_q31_q15`:

Performs an 'Arithmetic' RIGHT Shift on a 32-bit input by 'b' bit positions. For positive shift directions ($b > 0$), 'b' Lsb-bits are shifted out to the right and 'b' sign-extended Msb-bits fill in From the left. For negative shift directions ($b < 0$), 'b' Lsb's are shifted to the LEFT with 0's filling in the empty lsb position. The left shifting causes 'b' Msb-bits to fall off to the left, saturation is applied to any shift left value that overflows. This function calls the left-shift function to perform any 32-bit left shifts. This function does not provide any status-type information to indicate when overflows occur.

positive value: # of bits to right shift (sign extend) negative value: # of bits to left shift (zeros inserted at LSB's)

This function relates to the ETSI `L_shr` function.

Parameters

Parameters	Description
q31 a	32-bit signed integer value to be shifted
i16 b	16-bit signed integer shift index

libq_q31_ShiftRightRound_q31_i16 Function

'Arithmetic' RIGHT Shift on a 32-bit value

File

[libq_C.h](#)

C

```
q31 libq_q31_ShiftRightRound_q31_i16(q31, i16);
```

Returns

q31 result - Arithmetically shifted 32-bit signed integer output

Description

Function `libq_q31_ShiftRightRound_q31_q15`:

Performs an 'Arithmetic' RIGHT Shift on a 32-bit input by 'b' bits with Rounding applied. The rounding occurs before any shift by adding a bit weight of "1/2 Lsb", where the "Lsb" is the Ending (shifted) Lsb. For example: The initial `Bit#(i+b)` is after the right shift `Bit#(i)`, so the rounding bit weight is `Bit#(i+b-1)`. Rounding does not occur on left shifts, when b is negative. After rounding, this function calls the right-shift function to perform the actual 32-bit right shift. For positive shift directions ($b > 0$), 'b' Lsb-bits are shifted out to the right and 'b' sign-extended Msb-bits fill in From the left. For negative shift directions ($b < 0$), 'b' Lsb's are shifted to the LEFT with 0's filling in the empty lsb position. The left shifting causes 'b' Msb-bits to fall off to the left, saturation is applied to any shift left value that overflows. This function calls the left-shift function to perform the actual 32-bit left shift. This function does not provide any status-type flag to indicate occurrence of overflow.

positive value: # of bits to right shift (sign extend) negative value: # of bits to left shift (zeros inserted at LSB's)

This function relates to the ETSI `L_shr_r` function.

Parameters

Parameters	Description
q31 a	32-bit signed integer value to be shifted
i16 b	16-bit signed integer shift index

libq_q31_Sub_q31_q31 Function

Subtract two 32-bit 2s-complement fractional values

File

[libq_C.h](#)

C

```
q31 libq_q31_sub_q31_q31(q31, q31);
```

Returns

q31 result a-b on range: MINFRACT31 <= result <= MAXFRACT31

Description

Function `libq_q31_Sub_q31_q31`:

Subtract two 32-bit 2s-complement fractional (`op1 - op2`) to produce a 16-bit 2s-complement fractional difference result with saturation. The saturation is for handling the overflow/underflow cases, where the result is set to `MAX32` when an overflow occurs and the result is set to `MIN32` when an underflow occurs. This function does not produce any status flag to indicate when an overflow or underflow has occurred. It is assumed that the binary point is in exactly the same bit position for both 16-bit inputs and the resulting 16-bit output. This function relates to the ETSI sub function.

Data Types and Constants

q15 Type

File

[libq_C.h](#)

C

```
typedef int16_t q15;
```

Description

q15 n.n (signed)

q31 Type

File

[libq_C.h](#)

C

```
typedef int32_t q31;
```

Description

q31 n.n (signed)

q63 Type

File

[libq_C.h](#)

C

```
typedef int64_t q63;
```


Description

Q63 n.n (signed)

LIBQ_C_H_Macro**File**

[libq_C.h](#)

C

```
#define _LIBQ_C_H_
```

Description

This is macro `_LIBQ_C_H_`.

BITMASKFRACT16 Macro**File**

[libq_C.h](#)

C

```
#define BITMASKFRACT16 (~((uint16_t)0x0)) /* Bit Mask for 16 */
```

Description

Bit Mask for 16

BITMASKFRACT32 Macro**File**

[libq_C.h](#)

C

```
#define BITMASKFRACT32 (~((uint32_t)0x0L)) /* Bit Mask for 32 */
```

Description

Bit Mask for 32

F12Fract16 Macro

Converts floating point constant value to fractional 16-bit value

File

[libq_C.h](#)

C

```
#define F12Fract16(value) \
( \
(q15)F12Fxpnt((value), 15) \
)
```

Returns

`q15` - Equivalent fractional value

Description

Function `F12Fract16`:

Converts floating point constant value to fractional 16-bit value with rounding.

Parameters

Parameters	Description
value	Floating point value constant expression to convert.

F12Fract32 Macro

Converts floating point constant value to fractional 32-bit value

File

[libq_C.h](#)

C

```
#define F12Fract32(value) \
( \
    (q31)F12FxpPnt((value), 31) \
)
```

Returns

Equivalent fractional value as AtiFract32.

Description

Function F12Fract32:

Converts floating point constant value to fractional 32-bit value with rounding.

Parameters

Parameters	Description
value	Floating point value constant expression to convert.

F12FxpPnt Macro

Converts floating point constant value to fixed/fractional value

File

[libq_C.h](#)

C

```
#define F12FxpPnt(value, bits) \
( \
    ((double)(value)*((double)(1UL<<(bits)))) + \
    (((double)((double)(value) >= 0.0)) - ((double)0.5)) \
)
```

Returns

double - Equivalent floating point value

Description

Function F12FxpPnt:

Converts floating point constant value to fixed/fractional value of specified precision with rounding.

Parameters

Parameters	Description
value	Floating point value constant expression to convert.
int_t bits	Number of fractional bits.

F12FxpPnt16 Macro

Converts floating point constant value to fixed point 16-bit value

File

[libq_C.h](#)

C

```
#define F12FxpPnt16(value, bits) \
( \
    (q15)F12FxpPnt((value), (bits)) \
)
```

)

Returns

q15 - Equivalent fixed point value

Description

Function F12FxpPnt16:

Converts floating point constant value to fixed point 16-bit value with rounding.

Parameters

Parameters	Description
value	Floating point value constant expression to convert. int_t bits -- Number of fractional bits.

F12FxpPnt32 Macro

Converts floating point constant value to fixed point 32-bit value

File

libq_C.h

C

```
#define F12FxpPnt32(value, bits) \
( \
(q31)F12FxpPnt((value), (bits)) \
)
```

Returns

q31 - Equivalent fixed point value

Description

Function F12FxpPnt32:

Converts floating point constant value to fixed point 32-bit value with rounding.

Parameters

Parameters	Description
value	Floating point value constant expression to convert.
int_t bits	Number of fractional bits.

F12Int16 Macro

Converts floating point constant expression to 16-bit integer value

File

libq_C.h

C

```
#define F12Int16(value) ((int16_t)((double)(value)))
```

Returns

int_t - Calculated value.

Description

Function F12Int16:

Converts floating point constant expression to 16-bit integer value Conversion is safe for compilers which assign fractional data type to constants with decimal point

Parameters

Parameters	Description
value	Floating point value constant expression to convert.

F12Int32 Macro

Converts floating point constant expression to 32-bit integer value

File

[libq_C.h](#)

C

```
#define F12Int32(value) ((int32_t)((double)(value)))
```

Returns

int_t - Calculated integer value.

Description

Function F12Int32:

Converts floating point constant expression to 32-bit integer value Conversion is safe for compilers which assign fractional data type to constants with decimal point

Parameters

Parameters	Description
value	Floating point value constant expression to convert.

FrMax Macro

File

[libq_C.h](#)

C

```
#define FrMax(a,b) ((a)>(b)?(a):(b))
```

Description

find the maximum of two numbers

FrMin Macro

File

[libq_C.h](#)

C

```
#define FrMin(a,b) ((a)<(b)?(a):(b))
```

Description

find the minimum of two numbers

LOG102Q5D11 Macro

File

[libq_C.h](#)

C

```
#define LOG102Q5D11 F12FxpPnt16(0.301029996,11)
```

Description

log10(2) scaled to Q5.11 format

MAXFRACT16 Macro

File

[libq_C.h](#)

C

```
#define MAXFRACT16 MAXINT16 /* +0.999969 */
```

Description

0.999969

MAXFRACT32 Macro

File

[libq_C.h](#)

C

```
#define MAXFRACT32 MAXINT32 /* +0.9999999995 */
```

Description

0.9999999995

MAXINT16 Macro

File

[libq_C.h](#)

C

```
#define MAXINT16 ((int16_t) 0x7fff)
```

Description

Maximum and minimum values for 16-bit data types.

MAXINT32 Macro

File

[libq_C.h](#)

C

```
#define MAXINT32 ((int32_t)0x7fffffffL)
```

Description

This is macro MAXINT32.

MAXPFLOAT32 Macro

File

[libq_C.h](#)

C

```
#define MAXPFLOAT32 {MAXFRACT16, MAXINT16}
```

Description

minimum and maximum definitions for FX floating point data types

MINFRACT16 Macro

File

libq_C.h

C

```
#define MINFRACT16 MININT16 /* -1.000000 */
```

Description

1.000000

MINFRACT32 Macro

File

libq_C.h

C

```
#define MINFRACT32 MININT32 /* -1.0000000000 */
```

Description

1.0000000000

MININT16 Macro

File

libq_C.h

C

```
#define MININT16 ((int16_t) (-MAXINT16 - 1))
```

Description

This is macro MININT16.

MININT32 Macro

File

libq_C.h

C

```
#define MININT32 ((int32_t) (-MAXINT32 - 1))
```

Description

This is macro MININT32.

MINPFLOAT32 Macro

File

libq_C.h

C

```
#define MINPFLOAT32 {MINFRACT16, MAXINT16}
```

Description

This is macro MINPFLOAT32.

MSBBITFRACT16 Macro

File

libq_C.h

C

```
#define MSBBITFRACT16 MININT16          /* 16-bit Sign Bit */
```

Description

16-bit Sign Bit

MSBBITFRACT32 Macro

File

libq_C.h

C

```
#define MSBBITFRACT32 MININT32        /* 32-bit Sign Bit */
```

Description

32-bit Sign Bit

NINETYQ10D22 Macro

File

libq_C.h

C

```
#define NINETYQ10D22 F12FxPnt32(90,22) /* Ninety degrees scaled to Q10d22*/
```

Description

Ninety degrees scaled to Q10d22

NINETYQ10D6 Macro

File

libq_C.h

C

```
#define NINETYQ10D6 F12FxPnt16(90,6)  /* Ninety degrees scaled to Q10d6 */
```

Description

Ninety degrees scaled to Q10d6

NORMNEGFRACT16 Macro

File

libq_C.h

C

```
#define NORMNEGFRACT16 ((q15)0xc000)  /* Max -val for 16 */
```

Description

Max -val for 16

NORMNEGFRACT32 Macro

File

[libq_C.h](#)

C

```
#define NORMNEGFRACT32 ((q31)0xc0000000L)    /* Max -val for 32 */
```

Description

Max -val for 32

NORMPOSFRACT16 Macro

File

[libq_C.h](#)

C

```
#define NORMPOSFRACT16 ((q15)0x4000)        /* Min +val for 16 */
```

Description

Min +val for 16

NORMPOSFRACT32 Macro

File

[libq_C.h](#)

C

```
#define NORMPOSFRACT32 ((q31)0x40000000L)  /* Min +val for 32 */
```

Description

Min +val for 32

NUMBITSFRACT16 Macro

File

[libq_C.h](#)

C

```
#define NUMBITSFRACT16 ((int16_t)0x010)    /* Num of bits 16 */
```

Description

Num of bits 16

NUMBITSFRACT32 Macro

File

[libq_C.h](#)

C

```
#define NUMBITSFRACT32 ((int16_t)0x020)    /* Num of bits 32 */
```

Description

Num of bits 32

ONEEIGHTYQ10D22 Macro

File

[libq_C.h](#)

C

```
#define ONEEIGHTYQ10D22 F12FxPnt32(180,22) /* 180 degrees scaled to Q10d22 */
```

Description

180 degrees scaled to Q10d22

ONEEIGHTYQ10D6 Macro

File

[libq_C.h](#)

C

```
#define ONEEIGHTYQ10D6 F12FxPnt16(180,6) /* 180 degrees scaled to Q10d6 */
```

Description

180 degrees scaled to Q10d6

ROUNDFRACT32 Macro

File

[libq_C.h](#)

C

```
#define ROUNDFRACT32 ((q31)0x00008000L) /* Rounding value */
```

Description

Rounding value

THREESIXTYQ10D22 Macro

File

[libq_C.h](#)

C

```
#define THREESIXTYQ10D22 F12FxPnt32(360,22) /* 360 degrees scaled to Q10d22 */
```

Description

360 degrees scaled to Q10d22

THREESIXTYQ10D6 Macro

File

[libq_C.h](#)

C

```
#define THREESIXTYQ10D6 F12FxPnt16(360,6) /* 360 degrees scaled to Q10d6 */
```

Description

360 degrees scaled to Q10d6

TWOSEVENTYQ10D22 Macro

File

[libq_C.h](#)

C

```
#define TWOSEVENTYQ10D22 F12FxPnt32(270,22) /* 270 degrees scaled to Q10d22 */
```

Description

270 degrees scaled to Q10d22

TWOSEVENTYQ10D6 Macro

File

[libq_C.h](#)

C

```
#define TWOSEVENTYQ10D6 F12FxPnt16(270,6) /* 270 degrees scaled to Q10d6 */
```

Description

270 degrees scaled to Q10d6

UNITYFLOAT Macro

File

[libq_C.h](#)

C

```
#define UNITYFLOAT 1.0
```

Description

This is macro UNITYFLOAT.

FxQFloat32 Variable

File

[libq_C.h](#)

C

```
struct {  
    q15 man;  
    i16 exp;  
} FxQFloat32;
```

Description

FxQFloat32 pseudo floating point type (limited floating point)

Remarks

Extended FxQflExt32 used with f2Qfloat32)

Exponent16ToQFloat32 Macro

Converts a power of 2 16-bit integer to 32-bit floating point value.

File

[libq_C.h](#)

C

```
#define Exponent16ToQFloat32(value) \
```

```
( \
  (((value) < 0) ? ((float)1.0/(float)(1UL<<(-(value)))): \
  ((float)(1UL<<(value)))) \
)
```

Returns

32-bit floating point value of 2^n .

Description

Function Exponent16ToQFloat32

Converts a power of 2 16-bit integer to 32-bit floating point value.

Parameters

Parameters	Description
int16_t	16-bit integer number for power (expected to be a variable).

FI2QFloat32 Macro

Converts a decimal floating-point constant expression to pseudo float

File

[libq_C.h](#)

C

```
#define FI2QFloat32(mantissa, exponent) \
{ \
  F12Fract16(mantissa), \
  F12Int16(exponent) \
}
```

Returns

Calculated integer value.

Description

Function FI2QFloat32:

Converts a decimal floating-point constant expression to pseudo float

Parameters

Parameters	Description
mantissa	Floating-point value constant expression to convert to the mantissa portion of the floating-point number.
exponent	Integer value constant expression to convert to the exponent portion of the floating-point number.

i16 Type

File

[libq_C.h](#)

C

```
typedef int16_t i16;
```

Description

Q16d0

Files

Files

Name	Description
libq_C.h	C-code fixed point math functions.

Description

This section lists the source and header files used by the LibQ Fixed-Point 'C' Math Library.

libq_C.h

C-code fixed point math functions.

Functions

	Name	Description
⇒	Fx16Norm	Normalize the 16-bit fractional value.
⇒	Fx32Norm	Normalize the 32-bit number.
⇒	libq_q15_Abs_q15	Saturated Absolute value.
⇒	libq_q15_Add_q15_q15	Add two 16-bit 2s-complement fractional values.
⇒	libq_q15_DivisionWithSaturation_q15_q15	Fractional division with saturation.
⇒	libq_q15_ExpAvg_q15_q15_q1d15	Exponential averaging
⇒	libq_q15_ExtractH_q31	Extracts upper 16 bits of input 32-bit fractional value.
⇒	libq_q15_ExtractL_q31	Extracts lower 16-bits of input 32-bit fractional value. Description Extracts lower 16-bits of input 32-bit fractional value and returns them as 16-bit fractional value. This is a bit-for-bit extraction of the bottom 16-bits of the 32-bit input. This function relates to the ETSI extract_l function.
⇒	libq_q15_MacR_q31_q15_q15	Multiply accumulate with rounding.
⇒	libq_q15_MsuR_q31_q15_q15	Multiply-Subtraction with rounding
⇒	libq_q15_Negate_q15	Negate 16-bit 2s-complement fractional value with saturation.
⇒	libq_q15_RoundL_q31	Rounds the lower 16-bits of the 32-bit fractional input.
⇒	libq_q15_ShiftLeft_q15_i16	'Arithmetic' Shift of the 16-bit input argument.
⇒	libq_q15_ShiftRight_q15_i16	'Arithmetic' RIGHT Shift on a 16-bit value.
⇒	libq_q15_ShiftRightRound_q15_i16	Performs an 'Arithmetic' RIGHT Shift on a 16-bit input.
⇒	libq_q15_Sub_q15_q15	Subtract two 16-bit 2s-complement fractional values
⇒	libq_q1d15_Sin_q10d6	Approximates the sine of an angle.
⇒	libq_q20d12_Sin_q20d12	3rd order Polynomial apprx. of a sine function
⇒	libq_q31_Abs_q31	Saturated Absolute value.
⇒	libq_q31_Add_q31_q31	Add two 32-bit 2s-complement fractional values.
⇒	libq_q31_DepositH_q15	Place 16 bits in the upper half of 32 bit word.
⇒	libq_q31_DepositL_q15	Place 16 bits in the lower half of 32 bit word.
⇒	libq_q31_Mac_q31_q15_q15	Multiply-Accumulate function WITH saturation
⇒	libq_q31_Msu_q31_q15_q15	L_msu(a,b,c)
⇒	libq_q31_Mult2_q15_q15	fractional multiplication of two 16-bit fractional values.
⇒	libq_q31_Multi_q15_q31	Implement 16 bit by 32 bit multiply.
⇒	libq_q31_Negate_q31	Negate 32-bit 2s-complement fractional value with saturation.
⇒	libq_q31_ShiftLeft_q31_i16	'Arithmetic' Shift of the 32-bit value.
⇒	libq_q31_ShiftRight_q31_i16	'Arithmetic' RIGHT Shift on a 32-bit value.
⇒	libq_q31_ShiftRightRound_q31_i16	'Arithmetic' RIGHT Shift on a 32-bit value
⇒	libq_q31_Sub_q31_q31	Subtract two 32-bit 2s-complement fractional values

Macros

	Name	Description
	_LIBQ_C_H_	This is macro _LIBQ_C_H_.
	BITMASKFRACT16	Bit Mask for 16
	BITMASKFRACT32	Bit Mask for 32
	Exponent16ToQFloat32	Converts a power of 2 16-bit integer to 32-bit floating point value.
	F12Fract16	Converts floating point constant value to fractional 16-bit value
	F12Fract32	Converts floating point constant value to fractional 32-bit value
	F12Fxpnt	Converts floating point constant value to fixed/fractional value
	F12Fxpnt16	Converts floating point constant value to fixed point 16-bit value
	F12Fxpnt32	Converts floating point constant value to fixed point 32-bit value
	F12Int16	Converts floating point constant expression to 16-bit integer value

FI2Int32	Converts floating point constant expression to 32-bit integer value
FI2QFloat32	Converts a decimal floating-point constant expression to pseudo float
FrMax	find the maximum of two numbers
FrMin	find the minimum of two numbers
LOG102Q5D11	log ₁₀ (2) scaled to Q5.11 format
MAXFRACT16	0.999969
MAXFRACT32	0.9999999995
MAXINT16	Maximum and minimum values for 16-bit data types.
MAXINT32	This is macro MAXINT32.
MAXPFLOAT32	minimum and maximum definitions for FX floating point data types
MINFRACT16	1.000000
MINFRACT32	1.0000000000
MININT16	This is macro MININT16.
MININT32	This is macro MININT32.
MINPFLOAT32	This is macro MINPFLOAT32.
MSBBITFRACT16	16-bit Sign Bit
MSBBITFRACT32	32-bit Sign Bit
NINETYQ10D22	Ninety degrees scaled to Q10d22
NINETYQ10D6	Ninety degrees scaled to Q10d6
NORMNEGFRACT16	Max -val for 16
NORMNEGFRACT32	Max -val for 32
NORMPOSFRACT16	Min +val for 16
NORMPOSFRACT32	Min +val for 32
NUMBITSFRACT16	Num of bits 16
NUMBITSFRACT32	Num of bits 32
ONEEIGHTYQ10D22	180 degrees scaled to Q10d22
ONEEIGHTYQ10D6	180 degrees scaled to Q10d6
ROUNDFRACT32	Rounding value
THREESIXTYQ10D22	360 degrees scaled to Q10d22
THREESIXTYQ10D6	360 degrees scaled to Q10d6
TWOSEVENTYQ10D22	270 degrees scaled to Q10d22
TWOSEVENTYQ10D6	270 degrees scaled to Q10d6
UNITYFLOAT	This is macro UNITYFLOAT.

Types

Name	Description
i16	Q16d0
q15	q15 n.n (signed)
q31	q31 n.n (signed)
q63	Q63 n.n (signed)

Variables

Name	Description
FxQFloat32	FxQFloat32 pseudo floating point type (limited floating point)

Description

The libq_c Fixed-Point Math Library provides fixed-point math functions written in C for portability between core processors.

Signed fixed point types (fractional Q types specified by Qn.m) are named as follows in the library names:

Qndm where:

- n is the number of data bits to the left of the radix point
- m is the number of data bits to the right of the radix point
- a signed bit is implied (unless stated otherwise)

For convenience, short names are also defined for arbitrary scaled fractional types:

[q15](#) is signed fractional 16 bit value [q31](#) is signed fractional 32 bit value [i16](#) is signed integer, i.e. Q16d0

In addition, A pseudo floating point 32 bit format ([FxQFloat32](#)) is defined that consists of 16 mantissa and a 16 bit exponent (base 2).

Functions in the library are prefixed with the type of the return value and followed by argument types (in order):

libq___: libq_q15_sin_Q2d13

For example, `libq_q1d15_Sin_q10d6` returns a Q1.15 value equal to the sine of an angle specified as a Q10.6 value (in degrees between 0 and 360)

Argument types do not always match the return type. Refer to the function prototype for a specification of its arguments.

In some cases, both the return type and the argument type are specified within the function name. For example,

For arbitrary scaled types (`q15`, `q16`, `q31`, and `q32`) the scaling of the result will depend on the function and the scaling of the arguments. For instance, `libq_q15_Add_q15_q15(a,b)` will return a scaled value type that is the two input types (which must have equivalent scaled value type).

Remarks

The `libq_c` functions do not correspond to the libq fixed-point library optimized for the microaptive core processor and written in asm.

Table of LIBQ_C math functions:

Sine: `libq_q1d15_Sin_q10d6` `libq_q20d12_Sin_q20d12`

Abs: `libq_q15_Abs_q15` `libq_q31_Abs_q31`

Negate: `libq_q15_Negate_q15` `libq_q31_Negate_q31`

Round: `libq_q15_RoundL_q31`

Deposit: `libq_q31_DepositH_q15` `libq_q31_DepositL_q15`

Extract: `libq_q15_ExtractH_q31` `libq_q15_ExtractL_q31`

Add: `libq_q15_Add_q15_q15` `libq_q31_Add_q31_q31`

Subtract: `libq_q15_Sub_q15_q15` `libq_q31_Sub_q31_q31`

Shift(Scale): `libq_q15_ShiftLeft_q15_q15` `libq_q31_ShiftLeft_q31_q15` `libq_q15_ShiftRight_q15_q15` `libq_q31_ShiftRight_q31_q15`
`libq_q15_ShiftRightRound_q15_q15` `libq_q31_ShiftRightRound_q31_q15`

Multiply: `libq_q31_Mult2_q15_q15` `libq_q15_MultiplyR2_q15_q15` (DEPRECATED - mis-spelled) `libq_q15_MultiplyR2_q15_q15`
`libq_q31_Multi_q15_q31`

Divide: `libq_q15_DivisionWithSaturation_q15_q15`

Multiply-Accumulate: `libq_q31_Mac_q31_q15_q15` `libq_q15_MacR_q31_q15_q15`

Multiply-Subtract: `libq_q31_Msu_q31_q15_q15` `libq_q15_MsuR_q31_q15_q15`

Exponential-Averaging: `libq_q15_ExpAvg_q15_q15_q1d15`

Table of LIBQ_C conversion functions:

Normalize Q value: `Fx16Norm` `Fx32Norm`

Float-to-Q value: `FI2Fract16` `FI2Fract32` `FI2FxPnt16` `FI2FxPnt32` `FI2FxPnt`

Float-To-Integer: `FI2Int16` `FI2Int32`

Float-To-FxQFloat32: `FI2QFloat32`

Exponent-To-Float: `Exponent16ToFloat32`

File Name

`libq_c.h`

Company

Microchip Technology Inc.

LibQ Fixed-Point Math Library

This topic describes the LibQ Fixed-Point Math Library.

Introduction

The LibQ Fixed-Point Math Library is available for the PIC32MZ family of microcontrollers. This library was created from optimized assembly routines written specifically for devices with microAptiv™ core features.

Description

The LibQ Fixed-Point Math Library simplifies writing fixed point algorithms, supporting Q15, Q31 and other 16-bit and 32-bit data formats. Using the simple, C callable functions contained in the library, fast fixed point mathematical operations can be easily executed. Fixed-point mathematical calculations may replace some functions implemented in the floating point library (`math.h`), depending on performance and resolution requirements.

Functions included in the LibQ library include capabilities for trigonometric, power and logarithms, and data conversion. In many cases the functions are identical other than the precision of their operands and the corresponding value that they return.

These functions are implemented in efficient assembly, and generally tuned to optimize performance over code size. In some cases the library

breaks out functions that enable one to be optimized for accuracy, while another version is optimized for speed. These functions such as [_LIBQ_Q2_29_acos_Q31](#) and [_LIBQ_Q2_29_acos_Q31_Fast](#) are otherwise identical and can be used interchangeably. Each of these functions are typically used in computationally intensive real-time applications where execution time is a critical parameter.

Using the Library

This topic describes the basic architecture of the LibQ Fixed-Point Math Library and provides information and examples on its use.

Description

Interface Header File: [libq.h](#)

The interface to the LibQ Fixed-Point Math Library is defined in the [libq.h](#) header file. Any C language source (.c) file that uses the LibQ Fixed-Point Library should include [libq.h](#).

Library File:

The LibQ Fixed-Point Math Library archive (.a) file is installed with MPLAB Harmony.

Library Overview

The LibQ Fixed-Point Math Library contains functions for manipulating Q15, Q31 and other intermediate integer representations of real numbers. The [Library Interface](#) section details the operation of the data formats and explains each function in detail.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the DSP Fixed-Point Math Library.

Library Interface Section	Description
Divide Functions	_Q16 fixed point divide function.
Square Root Functions	Square root of a positive _Q16 fixed point value function.
Log Functions	Log calculation functions.
Power Functions	Power calculation functions.
Exponential Functions	Exponential calculation functions.
Sine Functions	Sine calculation functions.
Cosine Functions	Cosine calculation functions.
Target Functions	Target calculation functions.
Arcsin Functions	Arcsin calculation functions.
Arccos Functions	Arccos calculation functions.
Arctan2 Functions	Arctan2 calculation functions.
Random Number Functions	_Q15 and _Q31 pseudo-random value functions.
Float Functions	Float conversion functions.
String Functions	ASCII to _Q15 conversions.

Signed fixed-point types are defined as follows:

Q_n_m where:

- n is the number of data bits to the left of the radix point
- m is the number of data bits to the right of the radix point
- a signed bit is implied

For convenience, short names are also defined:

Exact Name	Number of Bits Required	Short Name
_Q0_15	16	_Q15
_Q15_16	32	_Q16
_Q0_31	32	_Q31

Q_n_m numerical values are used by the library processing data as integers. In this format the n represents the number of integer bits, and the m represents the number of fractional bits. All values assume a sign bit in the most significant bit. The range of the numerical value therefore is:

$$-2^{(n-1)} \text{ to } [2^{(n-1)} - 2^{(-m)}]; \text{ with a resolution of } 2^{(-m)}.$$

A [_Q16](#) format number ([_Q15_16](#)) would range from -32768.0 (0x8000 0000) to 32767.99998474 with a precision of 0.000015259 (or 2^{-16}).

For example, a numerical representation of the number 3.14159 in [_Q2_13](#) notation would be:

$$3.14159 * 2^{13} = 25735.9 \Rightarrow 0x6488$$

And converting from the [_Q7_8](#) format with the value 0x1D89 would be:

$$0x1D89 / 2^8 = 7561 / 256 \Rightarrow 29.5316, \text{ accurate to } 0.00391$$

Functions in the library are prefixed with the type of the return value. For example, [_LIBQ_Q16Sqrt](#) returns a [_Q16](#) value equal to the square root of its argument. Argument types do not always match the return type. Refer to the function prototype for a specification of its arguments.

In cases where the return value is not a fixed-point type, the argument type is appended to the function name. For example, [_LIBQ_ToFloatQ31](#) accepts a type [_Q31](#) argument.

In some cases, both the return type and the argument type are specified within the function name. For example:


Function Name	Return Type	Argument Type
_LIBQ_Q15_sin_Q2_13	_Q15	_Q2_13
_LIBQ_Q31_sin_Q2_29	_Q31	_Q2_29

Table of Library Functions


Math Function	Function Definition
Divide	_Q16_LIBQ_Q16Div (_Q16 dividend, _Q16 divisor);
Square Root	_Q16_LIBQ_Q16Sqrt (_Q16 x);
Exponential	_Q16_LIBQ_Q16Exp (_Q16 x);
Log	_Q4_11_LIBQ_Q4_11_In_Q16 (_Q16 x); _Q3_12_LIBQ_Q3_12_log10_Q16 (_Q16 x); _Q5_10_LIBQ_Q5_10_log2_Q16 (_Q16 x);
Power	_Q16_LIBQ_Q16Power (_Q16 x, _Q16 y);
Sine	_Q15_LIBQ_Q15_sin_Q2_13 (_Q2_13 x); _Q31_LIBQ_Q31_sin_Q2_29 (_Q2_29 x);
Cosine	_Q15_LIBQ_Q15_cos_Q2_13 (_Q2_13 x); _Q31_LIBQ_Q31_cos_Q2_29 (_Q2_29 x);
Tangent	_Q7_8_LIBQ_Q7_8_tan_Q2_13 (_Q2_13 x); _Q16_LIBQ_Q16_tan_Q2_29 (_Q2_29 x);
Arcsin	_Q2_13_LIBQ_Q2_13_asin_Q15 (_Q15 x); _Q2_29_LIBQ_Q2_29_asin_Q31 (_Q31 x); _Q2_29_LIBQ_Q2_29_asin_Q31_Fast (_Q31 x);
Arccos	_Q2_13_LIBQ_Q2_13_acos_Q15 (_Q15 x); _Q2_29_LIBQ_Q2_29_acos_Q31 (_Q31 x); _Q2_29_LIBQ_Q2_29_acos_Q31_Fast (_Q31 x);
Arctan	_Q2_13_LIBQ_Q2_13_atan_Q7_8 (_Q7_8 x); _Q2_29_LIBQ_Q2_29_atan_Q16 (_Q16 x);
Arctan2	_Q2_13_LIBQ_Q2_13_atan2_Q7_8 (_Q7_8 y, _Q7_8 x); _Q2_29_LIBQ_Q2_29_atan2_Q16 (_Q16 y, _Q16 x);
Random Number	_Q15_LIBQ_Q15Rand (int64_t *pSeed); _Q31_LIBQ_Q31Rand (int64_t *pSeed);
Float	float _LIBQ_ToFloatQ31 (_Q31 x); float _LIBQ_ToFloatQ15 (_Q15 x); _Q31_LIBQ_Q31FromFloat (float x); _Q15_LIBQ_Q15FromFloat (float x);
String	void _LIBQ_ToStringQ15 (_Q15 x, char *s); _Q15_LIBQ_Q15FromString (char *s);

Library Interface




a) Divide Functions

	Name	Description
	_LIBQ_Q16Div	_Q16 fixed point divide.


b) Square Root Functions

	Name	Description
	_LIBQ_Q16Sqrt	Square root of a positive _Q16 fixed point value.


c) Log Functions

	Name	Description
	_LIBQ_Q3_12_log10_Q16	Calculates the value of $\text{Log}_{10}(x)$.
	_LIBQ_Q4_11_ln_Q16	Calculates the natural logarithm $\ln(x)$.
	_LIBQ_Q5_10_log2_Q16	Calculates the value of $\log_2(x)$.



d) Power Functions

	Name	Description
	_LIBQ_Q16Power	Calculates the value of x raised to the y power (x^y).



e) Exponential Functions

	Name	Description
	_LIBQ_Q16Exp	Calculates the exponential function e^x .



f) Sine Functions

	Name	Description
	_LIBQ_Q15_sin_Q2_13	Calculates the value of $\text{sine}(x)$.
	_LIBQ_Q31_sin_Q2_29	Calculates the value of $\text{sine}(x)$.




g) Cosine Functions

	Name	Description
	_LIBQ_Q15_cos_Q2_13	Calculates the value of $\text{cosine}(x)$.
	_LIBQ_Q31_cos_Q2_29	Calculates the value of $\text{cosine}(x)$.




h) Target Functions

	Name	Description
	_LIBQ_Q16_tan_Q2_29	Calculates the value of $\tan(x)$.
	_LIBQ_Q7_8_tan_Q2_13	Calculates the value of $\tan(x)$.


i) Arcsin Functions


	Name	Description
	_LIBQ_Q2_13_asin_Q15	Calculates the asin value of $\text{asin}(x)$.
	_LIBQ_Q2_29_asin_Q31	Calculates the value of $\text{asin}(x)$.
	_LIBQ_Q2_29_asin_Q31_Fast	Calculates the value of $\text{asin}(x)$. This function executes faster than the _LIBQ_Q2_29_asin_Q31 function, but is less precise.

j) Arccos Functions



	Name	Description
	_LIBQ_Q2_13_acos_Q15	Calculates the value of $\text{acos}(x)$.
	_LIBQ_Q2_29_acos_Q31	Calculates the value of $\text{acos}(x)$.
	_LIBQ_Q2_29_acos_Q31_Fast	Calculates the value of $\text{acos}(x)$. This function executes faster than _LIBQ_Q2_29_acos_Q31 but is less precise.

k) Arctan Functions



	Name	Description
	_LIBQ_Q2_13_atan_Q7_8	Calculates the value of $\text{atan}(x)$.

	_LIBQ_Q2_29_atan_Q16	Calculates the value of atan(x).
---	--------------------------------------	----------------------------------





I) Arctan2 Functions

	Name	Description
	_LIBQ_Q2_13_atan2_Q7_8	Calculates the value of atan2(y, x).
	_LIBQ_Q2_29_atan2_Q16	Calculates the value of atan2(y, x).



m) Random Number Functions

	Name	Description
	_LIBQ_Q15Rand	Generate a _Q15 random number.
	_LIBQ_Q31Rand	Generate a _Q31 random number.

n) Float Functions

	Name	Description
	_LIBQ_Q15FromFloat	Converts a float to a _Q15 value.
	_LIBQ_Q31FromFloat	Converts a float to a _Q31 value.
	_LIBQ_ToFloatQ15	Converts a _Q15 value to a float.
	_LIBQ_ToFloatQ31	Converts a _Q31 value to a float.

o) String Functions

	Name	Description
	_LIBQ_Q15FromString	ASCII to _Q15 conversion.
	_LIBQ_ToStringQ15	_Q15 to ASCII conversion.

p) Data Types and Constants

	Name	Description
	_Q15_MAX	Maximum value of _Q15 (~1.0)
	_Q15_MIN	Minimum value of _Q15 (-1.0)
	_Q16_MAX	Maximum value of _Q16 (~32768.0)
	_Q16_MIN	Minimum value of _Q16 (-32768.0)
	_Q2_13_MAX	Maximum value of _Q2_13 (~4.0)
	_Q2_13_MIN	Minimum value of _Q2_13 (-4.0)
	_Q2_29_MAX	Maximum value of _Q2_29 (~4.0)
	_Q2_29_MIN	Minimum value of _Q2_29 (-4.0)
	_Q3_12_MAX	Maximum value of _Q3_12 (~8.0)
	_Q3_12_MIN	Minimum value of _Q3_12 (-8.0)
	_Q31_MAX	Maximum value of _Q31 (~1.0)
	_Q31_MIN	Minimum value of _Q31 (-1.0)
	_Q4_11_MAX	Maximum value of _Q4_11 (~16.0)
	_Q4_11_MIN	Minimum value of _Q4_11 (-16.0)
	_Q5_10_MAX	Maximum value of _Q5_10 (~32.0)
	_Q5_10_MIN	Minimum value of _Q5_10 (-32.0)
	_Q7_8_MAX	Maximum value of _Q7_8 (~128.0)
	_Q7_8_MIN	Minimum value of _Q7_8 (-128.0)
	_Q0_15	1 sign bit, 15 bits right of radix
	_Q0_31	1 sign bit, 31 bits right of radix
	_Q15	Short name for _Q0_15
	_Q15_16	1 sign bit, 15 bits left of radix, 16 bits right of radix
	_Q16	Short name for _Q15_16
	_Q2_13	1 sign bit, 2 bits left of radix, 13 bits right of radix
	_Q2_29	1 sign bit, 2 bits left of radix, 29 bits right of radix
	_Q3_12	1 sign bit, 3 bits left of radix, 12 bits right of radix
	_Q31	Short name for _Q_0_31
	_Q4_11	1 sign bit, 4 bits left of radix, 11 bits right of radix
	_Q5_10	1 sign bit, 5 bits left of radix, 10 bits right of radix

_Q7_8	1 sign bit, 7 bits left of radix, 8 bits right of radix
_LIBQ_H	Guards against multiple inclusion

Description

This section describes the Application Programming Interface (API) functions, macros, and types of the LibQ Fixed Point Math Library. Refer to each section for a detailed description.

a) Divide Functions

LIBQ_Q16Div Function

[_Q16](#) fixed point divide.

File

[libq.h](#)

C

```
_Q16 _LIBQ_Q16Div(_Q16 dividend, _Q16 divisor);
```

Returns

[_Q16](#) quotient of the divide operation

Description

Function `_LIBQ_Q16Div`:

```
_Q16 _LIBQ_Q16Div (_Q16 dividend, _Q16 divisor);
```

Quotient (`_Q16`) = Dividend (`_Q16`) / Divisor (`_Q16`).

Remarks

The `_LIBQ_Q16Div` operation saturates its result.

Execution Time (cycles): 143 typical (80 to 244) Program Memory 204 bytes

Error ≤ 0.000015258789 (accurate to least significant `_Q16` bit within the non-saturated range)

Preconditions

Divisor must not equal 0.

Example

```
_Q16 quotient, dividend, divisor;

dividend = (_Q16)0x00010000; // 1
divisor  = (_Q16)0x00008000; // 0.5

quotient = _LIBQ_Q16Div (dividend, divisor);

// quotient now equals 2; i.e., (_Q16)0x00020000;
```

Parameters

Parameters	Description
dividend	The divide operation dividend (<code>_Q16</code>)
divisor	The divide operation divisor (<code>_Q16</code>)

b) Square Root Functions

LIBQ_Q16Sqrt Function

Square root of a positive `_Q16` fixed point value.

File

[libq.h](#)

C

```
_Q16 _LIBQ_Q16Sqrt(_Q16 x);
```

Returns

_LIBQ_Q16Sqrt returns the [_Q16](#) fixed point value which is the square root of the input parameter.

Description

Function `_LIBQ_Q16Sqrt`:

```
\_Q16 _LIBQ_Q16Sqrt(\_Q16 x);
```

Calculate the square root of a positive [_Q16](#) fixed point value, and return the [_Q16](#) result.

Remarks

Execution Time (cycles): 240 typical (104 to 258) Program Memory 152 bytes

Error <= 0.000015258789 (accurate to least significant [_Q16](#) bit)

Preconditions

The input value must be positive.

Example

```
\_Q16 squareRoot;

squareRoot = _LIBQ_Q16Sqrt((_Q16)0x01000000); // The square root of 256.0 is 16.0 (0x00100000)
squareRoot = _LIBQ_Q16Sqrt((_Q16)0x00004000); // The square root of 0.25 is 0.5 (0x00008000)
squareRoot = _LIBQ_Q16Sqrt((_Q16)0x5851f42d); // The square root of 22609.953125 is 150.366074 (0x00965db7)
```

Parameters

Parameters	Description
x	The _Q16 fixed point value input from which to find the square root.

c) Log Functions***[_LIBQ_Q3_12_log10_Q16](#) Function***

Calculates the value of $\text{Log}_{10}(x)$.

File

[libq.h](#)

C

```
\_Q3\_12 _LIBQ_Q3_12_log10_Q16(\_Q16 x);
```

Returns

`_LIBQ_Q3_12_log10_Q16` returns the [_Q3_12](#) fixed point result from the calculation $\text{log}_{10}(x)$.

Description

Function `_LIBQ_Q3_12_log10_Q16`:

```
\_Q3\_12 _LIBQ_Q3_12_log10_Q16 (\_Q16 x);
```

Calculates the $\text{log}_{10}(x)$, where $\text{log}_{10}(x) = \ln(x) * \text{log}_{10}(e)$. x is of type [_Q16](#) and must be positive. The resulting value is of type [_Q3_12](#).

Remarks

Execution Time (cycles): 301 typical (14 to 346) Program Memory 176 bytes

Error <= 0.000244140625 (accurate to least significant [_Q3_12](#) bit)

Preconditions

The input x must be positive.

Example

```
_Q3_12 resultLog10;

resultLog10 = _LIBQ_Q3_12_log10_Q16 ((_Q16)0x12ed7d91); // _LIBQ_Q3_12_log10_Q16(4845.490494) = 3.685303
(0x3aF7)
```

Parameters

Parameters	Description
x	The input value from which to calculate log10(x).

LIBQ_Q4_11_In_Q16 Function

Calculates the natural logarithm ln(x).

File

libq.h

C

```
_Q4_11 _LIBQ_Q4_11_ln_Q16(_Q16 x);
```

Returns

_LIBQ_Q4_11_In_Q16 returns the [_Q4_11](#) fixed point result from the calculation ln(x).

Description

Function _LIBQ_Q4_11_In_Q16:

[_Q4_11](#) _LIBQ_Q4_11_In_Q16 ([_Q16](#) x);

Calculates the natural logarithm ln(x). x is of type [_Q16](#) and must be positive. The resulting value is of type [_Q4_11](#).

Remarks

Execution Time (cycles): 301 typical (14 to 346) Program Memory 176 bytes

Error <= 0.00048828 (accurate to least significant [_Q4_11](#) bit)

Preconditions

The input x must be positive.

Example

```
_Q4_11 resultLN;

resultLN = _LIBQ_Q4_11_ln_Q16 ((_Q16)0x00004000); // _LIBQ_Q4_11_LN_Q16(0.250000) = -1.386230 (0xf4e9)
```

Parameters

Parameters	Description
x	The input value from which to calculate ln(x).

LIBQ_Q5_10_log2_Q16 Function

Calculates the value of log2(x).

File

libq.h

C

```
_Q5_10 _LIBQ_Q5_10_log2_Q16(_Q16 x);
```

Returns

_LIBQ_Q5_10_log2_Q16 returns the [_Q5_10](#) fixed point result from the calculation log2(x).

Description

Function _LIBQ_Q5_10_log2_Q16:

[_Q5_10](#) _LIBQ_Q5_10_log2_Q16 ([_Q16](#) x);

Calculates the $\log_2(x)$, where $\log_2(x) = \ln(x) * \log_2(e)$. x is of type `_Q16` and must be positive. The resulting value is of type `_Q5_10`.

Remarks

Execution Time (cycles): 227 typical (14 to 268) Program Memory 164 bytes

Error ≤ 0.0009765625 (accurate to least significant `_Q5_10` bit)

Preconditions

The input x must be positive.

Example

```
_Q5_10 resultLog2;
```

```
resultLog2 = _LIBQ_Q5_10_log2_Q16 ((_Q16)0x40000000); // _LIBQ_Q5_10_log2_Q16(16384.000000) = 14.000000
(0x3800)
```

Parameters

Parameters	Description
x	The input value from which to calculate $\log_2(x)$.

d) Power Functions

LIBQ_Q16Power Function

Calculates the value of x raised to the y power (x^y).

File

[libq.h](#)

C

```
_Q16 _LIBQ_Q16Power(_Q16 x, _Q16 y);
```

Returns

`_LIBQ_Q16Power` returns the `_Q16` fixed point result from the calculation x raised to the y .

Description

Function `_LIBQ_Q16Power`:

```
_Q16 _LIBQ_Q16Power (_Q16 x, _Q16 y);
```

Calculates the x raised to the y power. Both x and y are of type `_Q16`. x must be positive. The calculation will saturate if the resulting value is outside the range of the `_Q16` representation.

Remarks

Execution Time (cycles): 882 typical (586 to 1042) Program Memory 1038 bytes

Error ≤ 0.000015258789 (accurate to least significant `_Q16` bit within the non-saturated range)

Preconditions

x must be positive.

Example

```
_Q16 resultPower;
```

```
resultPower = _LIBQ_Q16Power ((_Q16)0x00020000, (_Q16)0xffff0000); // _LIBQ_Q16Power(2.000000, -1.000000)
= 0.500000 (0x00008000)
```

Parameters

Parameters	Description
x	The <code>_Q16</code> input value x from which to calculate x raised to the y .
y	The <code>_Q16</code> input value y from which to calculate x raised to the y .

e) Exponential Functions

LIBQ_Q16Exp Function

Calculates the exponential function e^x .

File

[libq.h](#)

C

```
_Q16 _LIBQ_Q16Exp(_Q16 x);
```

Returns

_LIBQ_Q16Exp returns the [_Q16](#) fixed point result from the calculation e^x .

Description

Function [_LIBQ_Q16Exp](#):

```
\_Q16 \_LIBQ\_Q16Exp(\_Q16 x);
```

Calculates the exponential function e^x . The calculation will saturate if the resulting value is outside the range of the [_Q16](#) representation. For $x > 10.3972015380859375$, the resulting value will be saturated to 0x7ffffff. For $x < -10.3972015380859375$ the resulting value will be saturated to 0.

Remarks

The function [_LIBQ_Q16Div](#) is called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 170 typical (18 to 292) Program Memory 446 bytes

Error ≤ 0.000015258789 (accurate to least significant [_Q16](#) bit within the non-saturated range)

Preconditions

None.

Example

```
_Q16 expResult;

expResult = _LIBQ_Q16Exp((_Q16)0x00010000); // \_LIBQ\_Q16Exp(1.000000) = 2.718277 (0x0002b7e1)
```

Parameters

Parameters	Description
x	The exponent value

f) Sine Functions

LIBQ_Q15_sin_Q2_13 Function

Calculates the value of $\sin(x)$.

File

[libq.h](#)

C

```
_Q15 _LIBQ_Q15_sin_Q2_13(_Q2_13 x);
```

Returns

_LIBQ_Q15_sin_Q2_13 returns the [_Q15](#) fixed point result from the calculation $\sin(x)$.

Description

Function [_LIBQ_Q15_sin_Q2_13](#):

```
\_Q15 \_LIBQ\_Q15\_sin\_Q2\_13(\_Q2\_13 x);
```

Calculates the $\sin(x)$, where x is of type [_Q2_13](#) radians and the resulting value is of type [_Q15](#).

Remarks

Execution Time (cycles): 100 typical (100 to 102) Program Memory 220 bytes

Error <= 0.00003052 (accurate to least significant [_Q15](#) bit)

Preconditions

None.

Example

```
_Q15 resultSin;

resultSin = _LIBQ_Q15_sin_Q2_13 ((_Q2_13)0x4093); // _LIBQ_Q15_sin_Q2_13(2.017944) = 0.901672 (0x736a)
```

Parameters

Parameters	Description
x	The _Q2_13 input value from which to calculate sine(x).

LIBQ_Q31_sin_Q2_29 Function

Calculates the value of sine(x).

File

[libq.h](#)

C

```
_Q31 _LIBQ_Q31_sin_Q2_29(_Q2_29 x);
```

Returns

[_LIBQ_Q31_sin_Q2_29](#) returns the [_Q31](#) fixed point result from the calculation sine(x).

Description

Function [_LIBQ_Q31_sin_Q2_29](#):

```
\_Q31 \_LIBQ\_Q31\_sin\_Q2\_29 (\_Q2\_29 x);
```

Calculates the sine(x), where x is of type [_Q2_29](#) radians and the resulting value is of type [_Q31](#).

Remarks

Execution Time (cycles): 246 typical (244 to 266) Program Memory 598 bytes

Error <= 0.00000000047 (accurate to least significant [_Q31](#) bit)

Preconditions

None.

Example

```
_Q31 resultSin;

resultSin = _LIBQ_Q31_sin_Q2_29 ((_Q2_29)0x5a637cfe); // _LIBQ_Q31_sin_Q2_29( 2.824644562) = 0.311668121 (0x27e4bdb1)
```

Parameters

Parameters	Description
x	The _Q2_29 input value from which to calculate sine(x).

g) Cosine Functions

LIBQ_Q15_cos_Q2_13 Function

Calculates the value of cosine(x).

File

[libq.h](#)

C

```
_Q15 _LIBQ_Q15_cos_Q2_13(_Q2_13 x);
```


Returns

`_LIBQ_Q15_cos_Q2_13` returns the `_Q15` fixed point result from the calculation `cosine(x)`.

Description

Function `_LIBQ_Q15_cos_Q2_13`:

`_Q15 _LIBQ_Q15_cos_Q2_13 (_Q2_13 x)`;

Calculates the `cosine(x)`, where `x` is of type `_Q2_13` radians and the resulting value is of type `_Q15`.

Remarks

Execution Time (cycles): 102 cycles Program Memory 224 bytes

Error ≤ 0.00003052 (accurate to least significant `_Q15` bit)

Preconditions

None

Example

```
_Q15 resultCos;
```

```
resultCos = _LIBQ_Q15_cos_Q2_13 ((_Q2_13)0x2171); // _LIBQ_Q15_cos_Q2_13(1.045044) = 0.501862 (0x403d)
```

Parameters

Parameters	Description
<code>x</code>	The <code>_Q2_13</code> input value from which to calculate <code>cosine(x)</code> .

LIBQ_Q31_cos_Q2_29 Function

Calculates the value of `cosine(x)`.

File

`libq.h`

C

```
_Q31 _LIBQ_Q31_cos_Q2_29(_Q2_29 x);
```

Returns

`_LIBQ_Q31_cos_Q2_29` returns the `_Q31` fixed point result from the calculation `sine(x)`.

Description

Function `_LIBQ_Q31_cos_Q2_29`:

`_Q31 _LIBQ_Q31_cos_Q2_29 (_Q2_29 x)`;

Calculates the `cosine(x)`, where `x` is of type `_Q2_29` radians and the resulting value is of type `_Q31`.

Remarks

Execution Time (cycles): 265 typical (22 to 288) Program Memory 746 bytes

Error ≤ 0.0000000047 (accurate to least significant `_Q31` bit)

Preconditions

None.

Example

```
_Q31 resultCos;
```

```
resultCos = _LIBQ_Q31_cos_Q2_29 ((_Q2_29)0x07e2e1c2); // _LIBQ_Q31_cos_Q2_29( 0.246445540) = 0.969785686 (0x7c21eff7)
```

Parameters

Parameters	Description
<code>x</code>	The <code>_Q2_29</code> input value from which to calculate <code>cosine(x)</code> .

h) Target Functions

LIBQ_Q16_tan_Q2_29 Function

Calculates the value of $\tan(x)$.

File

[libq.h](#)

C

```
_Q16 LIBQ_Q16_tan_Q2_29(_Q2_29 x);
```

Returns

`_LIBQ_Q16_tan_Q2_29` returns the `_Q16` fixed point result from the calculation $\tan(x)$. The resulting value is saturated.

Description

Function `_LIBQ_Q16_tan_Q2_29`:

```
_Q16 LIBQ_Q16_tan_Q2_29 (_Q2_29 x);
```

Calculates the $\tan(x)$, where x is of type `_Q2_29` radians and the resulting value is of type `_Q16`.

Remarks

The functions `_LIBQ_Q31_sin_Q2_29`, `_LIBQ_Q31_cos_Q2_29`, and `_LIBQ_Q16Div` are called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 703 typical (22 to 796) Program Memory 88 bytes

Error ≤ 0.000015259 (accurate to least significant `_Q16` bit for the input range $-1.568 \dots 1.568$) Error rises from 0.0 to 0.065 for the input range $-1.568 \dots -1.570765808$ and $1.568 \dots 1.570765808$)

Preconditions

None

Example

```
_Q16 resultTan;

resultTan = _LIBQ_Q16_tan_Q2_29 ((_Q2_29)0x16720c36); // _LIBQ_Q16_tan_Q2_29( 0.701421838 ) = 0.844726562
(0x0000d840)
```

Parameters

Parameters	Description
<code>x</code>	The <code>_Q2_29</code> input value from which to calculate $\tan(x)$.

LIBQ_Q7_8_tan_Q2_13 Function

Calculates the value of $\tan(x)$.

File

[libq.h](#)

C

```
_Q7_8 LIBQ_Q7_8_tan_Q2_13(_Q2_13 x);
```

Returns

`_LIBQ_Q7_8_tan_Q2_13` returns the `_Q7_8` fixed point result from the calculation $\tan(x)$.

Description

Function `_LIBQ_Q7_8_tan_Q2_13`:

```
_Q7_8 LIBQ_Q7_8_tan_Q2_13 (_Q2_13 x);
```

Calculates the $\tan(x)$, where x is of type `_Q2_13` radians and the resulting value is of type `_Q7_8`.

Remarks

Execution Time (cycles): 288 typical (18 to 346) Program Memory 980 bytes
 Error <= 0.00390625 (accurate to least significant [_Q7_8](#) bit)

Preconditions

None

Example

```
_Q7_8 resultTan;

resultTan = _LIBQ_Q7_8_tan_Q2_13 ((_Q2_13)0x2e20); // _LIBQ_Q7_8_tan_Q2_13(1.441406) = 7.683594 (0x07af)
```

Parameters

Parameters	Description
x	The _Q2_13 input value from which to calculate tan(x).

i) Arcsin Functions

LIBQ_Q2_13_asin_Q15 Function

Calculates the asin value of asin(x).

File

[libq.h](#)

C

```
_Q2_13 _LIBQ_Q2_13_asin_Q15(_Q15 x);
```

Returns

[_LIBQ_Q2_13_asin_Q15](#) returns the [_Q2_13](#) fixed point result from the calculation asin(x).

Description

Function [_LIBQ_Q2_13_asin_Q15](#):

```
\_Q2\_13 \_LIBQ\_Q2\_13\_asin\_Q15 (\_Q15 x);
```

Calculates asin(x), where x is of type [_Q15](#) and the resulting value is of type [_Q2_13](#). The output value will be radians in the range $\pi \geq \text{result} \geq -\pi$.

Remarks

The functions [_LIBQ_Q16Sqrt](#) and [_LIBQ_Q16Div](#) are called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 578 typical (22 to 656) Program Memory 336 bytes

Error <= 0.00012207 (accurate to least significant [_Q2_13](#) bit)

A higher resolution version of this function exists with equivalent performance, see [_LIBQ_Q2_29_asin_Q31_Fast](#)

Preconditions

None.

Example

```
_Q2_13 resultAsin;

resultAsin = _LIBQ_Q2_13_asin_Q15 ((_Q15)0x3231); // _LIBQ_Q2_13_asin_Q15(0.392120) = 0.402954 (0x0ce5)
```

Parameters

Parameters	Description
x	The _Q15 input value from which to calculate asin(x).

LIBQ_Q2_29_asin_Q31 Function

Calculates the value of asin(x).

File

libq.h

C

```
_Q2_29 _LIBQ_Q2_29_asin_Q31(_Q31 x);
```

Returns

_LIBQ_Q2_29_asin_Q31 returns the [_Q2_29](#) fixed point result from the calculation asin(x).

Description

Function `_LIBQ_Q2_29_asin_Q31`:

```
\_Q2\_29 _LIBQ_Q2_29_asin_Q31 (\_Q31 x);
```

Calculates the asin(x), where x is of type [_Q31](#) and the resulting value is of type [_Q2_29](#). The output value will be in radians the range $\pi \geq \text{result} \geq -\pi$.

Remarks

The functions `_LIBQ_Q2_29_asin_Q31_Fast` and `_LIBQ_Q31_sin_Q2_29` are called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 2525 typical (286 to 4330) Program Memory 138 bytes

Error ≤ 0.0000000019 (accurate to least significant [_Q2_29](#) bit for the range -0.9993..0.9993) Error ≤ 0.0000000346 (accurate to 5th least significant [_Q2_29](#) bit for the range -1.0 .. -0.9993 and 0.9993 .. 1.0)

A faster version of this function exists with modestly reduced accuracy, see [_LIBQ_Q2_29_asin_Q31_Fast](#)

Preconditions

None.

Example

```
\_Q2\_29 resultAsin;

resultAsin = _LIBQ_Q2_29_asin_Q31 ((\_Q31)0x7fe50658); // \_LIBQ\_Q2\_29\_asin\_Q31( 0.9991767816) =
1.5302172359 (0x30f78a23)
```

Parameters

Parameters	Description
x	The _Q31 input value from which to calculate asin(x).

LIBQ_Q2_29_asin_Q31_Fast Function

Calculates the value of asin(x). This function executes faster than the [_LIBQ_Q2_29_asin_Q31](#) function, but is less precise.

File

libq.h

C

```
_Q2_29 _LIBQ_Q2_29_asin_Q31_Fast(_Q31 x);
```

Returns

_LIBQ_Q2_29_asin_Q31_Fast returns the [_Q2_29](#) fixed point result from the calculation asin(x).

Description

Function `_LIBQ_Q2_29_asin_Q31_Fast`:

```
\_Q2\_29 _LIBQ_Q2_29_asin_Q31_Fast (\_Q31 x);
```

Calculates the asin(x), where x is of type [_Q31](#) and the resulting value is of type [_Q2_29](#). The output value will be in radians the range $\pi \geq \text{result} \geq -\pi$.

Remarks

Execution Time (cycles): 507 typical (22 to 1300) Program Memory 638 bytes

Error ≤ 0.000000911 (accurate to 9 least significant [_Q2_29](#) bits)

A higher resolution version of this function exists with reduced performance, see [_LIBQ_Q2_29_asin_Q31](#)

Preconditions

None.

Example

```

_Q2_29 resultAsin;

resultAsin = _LIBQ_Q2_29_asin_Q31_Fast ((_Q31)0x7fe50658); // _LIBQ_Q2_29_asin_Q31_Fast( 0.9991767816) =
1.5302172359 (0x30f78a23)

```

Parameters

Parameters	Description
x	The _Q31 input value from which to calculate asin(x).

j) Arccos Functions

LIBQ_Q2_13_acos_Q15 Function

Calculates the value of acos(x).

File

[libq.h](#)

C

```

_Q2_13 _LIBQ_Q2_13_acos_Q15(_Q15 x);

```

Returns

[_LIBQ_Q2_13_acos_Q15](#) returns the [_Q2_13](#) fixed point result from the calculation acos(x).

Description

Function [_LIBQ_Q2_13_acos_Q15](#):

[_Q2_13](#) [_LIBQ_Q2_13_acos_Q15](#) ([_Q15](#) x);

Calculates the acos(x), where x is of type [_Q15](#) and the resulting value is of type [_Q2_13](#). The output value will be radians in the range $\pi \geq \text{result} \geq -\pi$.

Remarks

The function [_LIBQ_Q2_13_asin_Q15](#) is called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 588 typical (32 to 666) Program Memory 24 bytes

Error ≤ 0.00012207 (accurate to least significant [_Q2_13](#) bit)

A higher precision function with equivalent performance exists, see [_LIBQ_Q2_29_acos_Q31_Fast](#)

Preconditions

None.

Example

```

_Q2_13 resultAcos;

resultAcos = _LIBQ_Q2_13_acos_Q15((_Q15)0x2993); // _LIBQ_Q2_13_acos_Q15(0.324799) = 1.239990 (0x27ae)

```

Parameters

Parameters	Description
x	The _Q15 input value from which to calculate acos(x).

LIBQ_Q2_29_acos_Q31 Function

Calculates the value of acos(x).

File

[libq.h](#)

C

```
_Q2_29 _LIBQ_Q2_29_acos_Q31(_Q31 x);
```

Returns

`_LIBQ_Q2_29_acos_Q31` returns the `_Q2_29` fixed point result from the calculation `acos(x)`.

Description

Function `_LIBQ_Q2_29_acos_Q31`:

```
_Q2_29 _LIBQ_Q2_29_acos_Q31 (_Q31 x);
```

Calculates the `acos(x)`, where `x` is of type `_Q31` and the resulting value is of type `_Q2_29`. The output value will be radians in the range $\pi \geq \text{result} \geq -\pi$.

Remarks

The functions `_LIBQ_Q2_29_asin_Q31_Fast` and `_LIBQ_Q31_cos_Q2_29` are called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 3370 typical (70 to 4824) Program Memory 142 bytes

Error ≤ 0.0000000019 (accurate to least significant `_Q2_29` bit for the range $-0.9993..0.9993$) Error ≤ 0.0000000355 (accurate to 5th least significant `_Q2_29` bit for the range $-1.0 .. -0.9993$ and $0.9993 .. 1.0$)

A similar function with higher performance and reduced precision exists, see `_LIBQ_Q2_29_acos_Q31_Fast`

Preconditions

None.

Example

```
_Q2_29 resultAcos;
```

```
resultAcos = _LIBQ_Q2_29_acos_Q31 ((_Q31)0xee63708c); // _LIBQ_Q2_29_acos_Q31(-0.1375903431) =
1.7088244837 (0x36aeb0af)
```

Parameters

Parameters	Description
<code>x</code>	The <code>_Q31</code> input value from which to calculate <code>acos(x)</code> .

LIBQ_Q2_29_acos_Q31_Fast Function

Calculates the value of `acos(x)`. This function executes faster than `_LIBQ_Q2_29_acos_Q31` but is less precise.

File

`libq.h`

C

```
_Q2_29 _LIBQ_Q2_29_acos_Q31_Fast(_Q31 x);
```

Returns

`_LIBQ_Q2_29_acos_Q31_Fast` returns the `_Q2_29` fixed point result from the calculation `acos(x)`.

Description

Function `_LIBQ_Q2_29_acos_Q31_Fast`:

```
_Q2_29 _LIBQ_Q2_29_acos_Q31_Fast (_Q31 x);
```

Calculates the `acos(x)`, where `x` is of type `_Q31` and the resulting value is of type `_Q2_29`. The output value will be radians in the range $\pi \geq \text{result} \geq -\pi$.

Remarks

The function `_LIBQ_Q2_29_asin_Q31_Fast` is called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 517 typical (32 to 1310) Program Memory 28 bytes

Error ≤ 0.0000000911 (accurate to 9 least significant `_Q2_29` bits)

A higher precision function with reduced performance exists, see `_LIBQ_Q2_29_acos_Q31`

Preconditions

None.

Example

```
_Q2_29 resultAcos;

resultAcos = _LIBQ_Q2_29_acos_Q31_Fast ((_Q31)0xee63708c); // _LIBQ_Q2_29_acos_Q31_Fast(-0.1375903431) =
1.7088244837 (0x36aeb0af)
```

Parameters

Parameters	Description
x	The _Q31 input value from which to calculate acos(x).

k) Arctan Functions***LIBQ_Q2_13_atan_Q7_8 Function***

Calculates the value of atan(x).

File

[libq.h](#)

C

```
_Q2_13 _LIBQ_Q2_13_atan_Q7_8(_Q7_8 x);
```

Returns

[_LIBQ_Q2_13_atan_Q7_8](#) returns the [_Q2_13](#) fixed point result from the calculation atan(x).

Description

Function [_LIBQ_Q2_13_atan_Q7_8](#):

```
\_Q2\_13 \_LIBQ\_Q2\_13\_atan\_Q7\_8 (\_Q7\_8 x);
```

Calculates the atan(x), where x is of type [_Q7_8](#) and the resulting value is of type [_Q2_13](#). The output value will be radians in the range $\pi \geq \text{result} \geq -\pi$.

Remarks

The function [_LIBQ_Q2_13_atan2_Q7_8](#) is called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 240 typical (202 to 256) Program Memory 16 bytes

Error ≤ 0.00012207 (accurate to least significant [_Q2_13](#) bit)

Preconditions

None.

Example

```
_Q2_13 resultAtan;

resultAtan = _LIBQ_Q2_13_atan_Q7_8 ((_Q7_8)0x0097); // _LIBQ_Q2_13_atan_Q7_8(0.589844) = 0.532959 (0x110e)
```

Parameters

Parameters	Description
x	The _Q7_8 input value from which to calculate atan(x).

LIBQ_Q2_29_atan_Q16 Function

Calculates the value of atan(x).

File

[libq.h](#)

C

```
_Q2_29 _LIBQ_Q2_29_atan_Q16(_Q16 x);
```

Returns

`_LIBQ_Q2_29_atan_Q16` returns the `_Q2_29` fixed point result from the calculation `atan(x)`.

Description

Function `_LIBQ_Q2_29_atan_Q16`:

`_Q2_29 _LIBQ_Q2_29_atan_Q16 (_Q16 x)`;

Calculates the `atan(x)`, where `x` is of type `_Q16` and the resulting value is of type `_Q2_29`. The output value will be radians in the range $\pi \geq \text{result} \geq -\pi$.

Remarks

The function `_LIBQ_Q2_29_atan2_Q16` is called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 354 typical (178 to 360) Program Memory 16 bytes

Error ≤ 0.000000003 (accurate within one least significant `_Q2_29` bit)

Preconditions

None.

Example

```
_Q2_29 resultAtan;
```

```
resultAtan = _LIBQ_Q2_29_atan_Q16 ((_Q16)0x00098b31); // _LIBQ_Q2_29_atan_Q16(9.543716) = 1.466396
(0x2eecb7ee)
```

Parameters

Parameters	Description
<code>x</code>	The <code>_Q16</code> input value from which to calculate <code>atan(x)</code> .

I) Arctan2 Functions

`_LIBQ_Q2_13_atan2_Q7_8` Function

Calculates the value of `atan2(y, x)`.

File

`libq.h`

C

```
_Q2_13 _LIBQ_Q2_13_atan2_Q7_8(_Q7_8 y, _Q7_8 x);
```

Returns

`_LIBQ_Q2_13_atan2_Q7_8` returns the `_Q2_13` fixed point result from the calculation `atan2(y, x)`.

Description

Function `_LIBQ_Q2_13_atan2_Q7_8`:

`_Q2_13 _LIBQ_Q2_13_atan2_Q7_8 (_Q7_8 y, _Q7_8 x)`;

Calculates the `atan2(y, x)`, where `y` and `x` are of type `_Q7_8` and the resulting value is of type `_Q2_13`. The output value will be radians in the range $\pi \geq \text{result} \geq -\pi$.

Remarks

The function `_LIBQ_Q16Div` is called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 220 typical (22 to 250) Program Memory 288 bytes

Error ≤ 0.00012207 (accurate to least significant `_Q2_13` bit)

Preconditions

None.

Example

```
_Q2_13 resultAtan2;
```



```
resultAtan2 = _LIBQ_Q2_13_atan2_Q7_8 ((_Q7_8)0x589d, (_Q7_8)0xf878); // _LIBQ_Q2_13_atan2_Q7_8(88.613281,
-7.531250) = 1.655518 (0x34fa)
```

Parameters

Parameters	Description
y	The _Q7_8 input value from which to calculate atan2(y, x).
x	The _Q7_8 input value from which to calculate atan2(y, x).

LIBQ_Q2_29_atan2_Q16 Function

Calculates the value of atan2(y, x).

File

[libq.h](#)

C

```
_Q2_29 _LIBQ_Q2_29_atan2_Q16(_Q16 y, _Q16 x);
```

Returns

[_LIBQ_Q2_29_atan2_Q16](#) returns the [_Q2_29](#) fixed point result from the calculation atan2(y, x).

Description

Function [_LIBQ_Q2_29_atan2_Q16](#):

```
\_Q2\_29 \_LIBQ\_Q2\_29\_atan2\_Q16 (\_Q16 y, \_Q16 x);
```

Calculates the atan(y, x), where y and x are of type [_Q16](#) and the resulting value is of type [_Q2_29](#). The output value will be radians in the range $\pi \geq \text{result} \geq -\pi$.

Remarks

The C function `__divdi3` is called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 348 typical (20 to 376) Program Memory 464 bytes

Error ≤ 0.000000003 (accurate within one least significant [_Q2_29](#) bit)

Preconditions

None.

Example

```
\_Q2\_29 resultAtan2;
```

```
resultAtan2 = _LIBQ_Q2_29_atan2_Q16 ((_Q16)0xf6276270, x(_Q16)0x34b4b4c0); //
_LIBQ_Q2_29_atan2_Q16(-2520.615479, 13492.706055) = -0.184684 (0xfa1710c7)
```

Parameters

Parameters	Description
y	The _Q16 input value from which to calculate atan2(y, x).
x	The _Q16 input value from which to calculate atan2(y, x).

m) Random Number Functions

LIBQ_Q15Rand Function

Generate a [_Q15](#) random number.

File

[libq.h](#)

C

```
\_Q15 _LIBQ_Q15Rand(int64_t * pSeed);
```

Returns

`_LIBQ_Q15Rand` returns a random `_Q15` value. `_LIBQ_Q15Rand` also updates the `int64_t *pSeed` value.

Description

Function `_LIBQ_Q15Rand`:

```
_Q15 _LIBQ_Q15Rand (int64_t *pSeed);
```

Generates a `_Q15` pseudo-random value based on the seed supplied as a parameter. The first time this function is called, the seed value must be supplied by the user; this initial seed value can either be constant or random, depending on whether the user wants to generate a repeatable or a non-repeatable pseudo-random sequence.

The function updates the `*pSeed` value each time it is called. The updated `*pSeed` value must be passed back to the function with each subsequent call.

Warning: The pseudo-random sequence generated by this function may be insufficient for cryptographic use.

Remarks

Execution Time (cycles): 32 Program Memory 92 bytes

Preconditions

None.

Example

```
// Initialize seed to a constant or random value
static int64_t randomSeed = 0xA71078BE72D4C1F1;

_Q15        randomValue;

randomValue = _LIBQ_Q15Rand(&randomSeed);
...
randomValue = _LIBQ_Q15Rand(&randomSeed);
```

Parameters

Parameters	Description
<code>pSeed</code>	A pointer to the seed value used by the function to generate a pseudo-random sequence.

LIBQ_Q31Rand Function

Generate a `_Q31` random number.

File

`libq.h`

C

```
_Q31 _LIBQ_Q31Rand(int64_t * pSeed);
```

Returns

`_LIBQ_Q31Rand` returns a pseudo-random `_Q31` value. `_LIBQ_Q31Rand` also updates the `int64_t *pSeed` value.

Description

Function `_LIBQ_Q31Rand`:

```
_Q31 _LIBQ_Q31Rand (int64_t *pSeed);
```

Generates a `_Q31` pseudo-random value based on the seed supplied as a parameter. The first time this function is called, the seed value must be supplied by the user; this initial seed value can either be constant or random, depending on whether the user wants to generate a repeatable or a non-repeatable pseudo-random sequence.

The function updates the `*pSeed` value each time it is called. The updated `*pSeed` value must be passed back to the function with each subsequent call.

Warning: The pseudo-random sequence generated by this function may be insufficient for cryptographic use.

Remarks

Execution Time (cycles): 32 Program Memory 88 bytes

Preconditions

None.

Example

```
// Initialize seed to a constant or random value
static int64_t randomSeed = 0x7F18BA710E72D4C1;

_Q31 randomValue;

randomValue = _LIBQ_Q31Rand(&randomSeed);
...
randomValue = _LIBQ_Q31Rand(&randomSeed);
```

Parameters

Parameters	Description
pSeed	A pointer to the seed value used by the function to generate a pseudo-random sequence.

n) Float Functions

LIBQ_Q15FromFloat Function

Converts a float to a [_Q15](#) value.

File

libq.h

C

```
_Q15 _LIBQ_Q15FromFloat(float x);
```

Returns

[_LIBQ_Q15FromFloat](#) returns the [_Q15](#) fixed point value corresponding to the floating point (float) input value.

Description

Function [_LIBQ_Q15FromFloat](#):

```
\_Q15 \_LIBQ\_Q15FromFloat(float x);
```

Converts a floating point value to a [_Q15](#) fixed point representation. The [_Q15](#) fixed point value is returned by the function. The conversion will saturate if the value is outside the range of the [_Q15](#) representation.

Remarks

The C library functions `__gesf2`, `__lesf2`, `__addsf3`, `__mulsf3`, and `__fixfsi` are called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 213 typical (158 to 224) Program Memory 96 bytes

Preconditions

None.

Example

```
_Q15 q15;

q15 = _LIBQ_Q15FromFloat((float)0.5); // q15 now equals (_Q15)0x4000

q15 = _LIBQ_Q15FromFloat((float)-1.0); // q15 now equals (_Q15)0x8000

q15 = _LIBQ_Q15FromFloat((float)-0.233828); // q15 now equals (_Q15)0xe212
```

Parameters

Parameters	Description
x	The float point value to convert to _Q15 fixed point

LIBQ_Q31FromFloat Function

Converts a float to a [_Q31](#) value.

File

libq.h

C

```
_Q31 _LIBQ_Q31FromFloat(float x);
```

Returns

_LIBQ_Q31FromFloat returns the [_Q31](#) fixed point value corresponding to the floating point (float) input value.

Description

Function `_LIBQ_Q31FromFloat`:

```
\_Q31 _LIBQ_Q31FromFloat(float x);
```

Converts a floating point value to a [_Q31](#) fixed point representation. The [_Q31](#) fixed point value is returned by the function. The conversion will saturate if the value is outside the range of the [_Q31](#) representation.

Remarks

The C library functions `__gesf2`, `__lesf2`, `__addsf3`, `__mulsf3`, and `__fixfsi` are called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 210 typical (158 to 214) Program Memory 100 bytes

Preconditions

None.

Example

```
_Q31 q31;

q31 = _LIBQ_Q31FromFloat((float)0.000008); // q31 now equals (_Q31)0x00004000

q31 = _LIBQ_Q31FromFloat((float)-1.0); // q31 now equals (_Q31)0x80000000

q31 = _LIBQ_Q31FromFloat((float)0.690001); // q31 now equals (_Q31)0x5851f400
```

Parameters

Parameters	Description
x	The floating point value to convert to _Q31 fixed point.

LIBQ_ToFloatQ15 Function

Converts a [_Q15](#) value to a float.

File

libq.h

C

```
float _LIBQ_ToFloatQ15(_Q15 x);
```

Returns

_LIBQ_ToFloatQ15 returns the floating point (float) value corresponding to the [_Q15](#) input value.

Description

Function `_LIBQ_ToFloatQ15`:

```
float _LIBQ_ToFloatQ15(_Q15 x);
```

Converts a [_Q15](#) fixed point value to a floating point representation. The floating point value is returned by the function.

Remarks

The C library functions `__floatsisf` and `__divsf3` are called by this routine and thus must be linked in to the executable image.

Execution Time (cycles): 158 typical (54 to 176) Program Memory 28 bytes

Preconditions

None.

Example

```
float f;

f = _LIBQ_ToFloatQ15((_Q15)0x4000); // f now equals 0.5

f = _LIBQ_ToFloatQ15((_Q15)0x8000); // f now equals -1.0

f = _LIBQ_ToFloatQ15((_Q15)0xb7ff); // f now equals -0.562531
```

Parameters

Parameters	Description
x	The _Q15 fixed point value to convert to float

LIBQ_ToFloatQ31 Function

Converts a **_Q31** value to a float.

File

libq.h

C

```
float _LIBQ_ToFloatQ31(_Q31 x);
```

Returns

_LIBQ_ToFloatQ31 returns the floating point (float) value corresponding to the **_Q31** input value.

Description

Function **_LIBQ_ToFloatQ31**:

```
float _LIBQ_ToFloatQ31(_Q31 x);
```

Converts a **_Q31** fixed point value to a floating point representation. The floating point value is returned by the function.

Remarks

The C library functions `__floatsisf` and `__divsf3` are called by this routine and thus must be linked in to the executable image.

Execution Time (cycles): 163 typical (54 to 176) Program Memory 28 bytes

Preconditions

None.

Example

```
float f;

f = _LIBQ_ToFloatQ31((_Q31)0x00004000); // f now equals 0.000008

f = _LIBQ_ToFloatQ31((_Q31)0x80000000); // f now equals -1.0

f = _LIBQ_ToFloatQ31((_Q31)0x5851f42d); // f now equals 0.690001
```

Parameters

Parameters	Description
x	The _Q31 fixed point value to convert to float

o) String Functions**LIBQ_Q15FromString Function**

ASCII to **_Q15** conversion.

File

libq.h

C

```
_Q15 _LIBQ_Q15FromString(char * s);
```

Returns

_LIBQ_Q15FromString returns the [_Q15](#) fixed point value represented by the input string.

Description

Function `_LIBQ_Q15FromString`:

```
\_Q15 _LIBQ_Q15FromString(char *s);
```

Convert an ASCII string into a [_Q15](#) fixed point value. The ASCII string must be in an -N.NNNNNN format. Leading spaces are ignored. The conversion stops at either the first non-conforming character in the string or the Null string terminator. There must be no spaces within the string value itself.

Remarks

Execution Time (cycles): 296 typical (28 to 346) Program Memory 172 bytes

Preconditions

None.

Example

```
_Q15 x;

x = _LIBQ_Q15FromString("0.125"); // x will equal 0.125 using
// an internal value of 0x1000

x = _LIBQ_Q15FromString("-1.0"); // x will equal -1.0 using
// an internal value of 0x8000

x = _LIBQ_Q15FromString("0.999969"); // x will equal 0.999969 using
// an internal value of 0x7FFF
```

Parameters

Parameters	Description
s	A pointer to the ASCII input string representing the _Q15 fixed point value.

LIBQ_ToStringQ15 Function

[_Q15](#) to ASCII conversion.

File

[libq.h](#)

C

```
void _LIBQ_ToStringQ15(_Q15 x, char * s);
```

Returns

An ASCII string that represents the [_Q15](#) fixed point value in -N.NNNNNN format. The output string will be terminated by a Null (0x00) character.

Description

Function `_LIBQ_ToStringQ15`:

```
void _LIBQ_ToStringQ15(\_Q15 x, char *s);
```

Convert a [_Q15](#) fixed point value to an ASCII string representation in a -N.NNNNNN format.

Remarks

Execution Time (cycles): 118 typical (28 to 132) Program Memory 200 bytes

Preconditions

The character string "s" must be at least 10 characters long, including the Null string terminator.

Example

```
char s[10];
```

```

_LIBQ_ToStringQ15((_Q15)0x1000, s);    // s will equal "0.125000"
_LIBQ_ToStringQ15((_Q15)0x8000, s);    // s will equal "-1.000000"
_LIBQ_ToStringQ15((_Q15)0x7FFF, s);    // s will equal "0.999969"

```

Parameters

Parameters	Description
x	The fixed point value to be converted into an ASCII string (_Q15)
s	A pointer to the output string of at least 10 characters

p) Data Types and Constants

[_Q15_MAX Macro](#)

File

[libq.h](#)

C

```
#define _Q15_MAX ((_Q15)0x7FFF)    // Maximum value of _Q15 (~1.0)
```

Description

Maximum value of [_Q15](#) (~1.0)

[_Q15_MIN Macro](#)

File

[libq.h](#)

C

```
#define _Q15_MIN ((_Q15)0x8000)    // Minimum value of _Q15 (-1.0)
```

Description

Minimum value of [_Q15](#) (-1.0)

[_Q16_MAX Macro](#)

File

[libq.h](#)

C

```
#define _Q16_MAX ((_Q16)0x7FFFFFFF)    // Maximum value of _Q16 (~32768.0)
```

Description

Maximum value of [_Q16](#) (~32768.0)

[_Q16_MIN Macro](#)

File

[libq.h](#)

C

```
#define _Q16_MIN ((_Q16)0x80000000)    // Minimum value of _Q16 (-32768.0)
```

Description

Minimum value of [_Q16](#) (-32768.0)

Q2_13_MAX Macro

File

libq.h

C

```
#define _Q2_13_MAX ((_Q2_13)0x7FFF) // Maximum value of _Q2_13 (~4.0)
```

Description

Maximum value of [_Q2_13](#) (~4.0)

Q2_13_MIN Macro

File

libq.h

C

```
#define _Q2_13_MIN ((_Q2_13)0x8000) // Minimum value of _Q2_13 (-4.0)
```

Description

Minimum value of [_Q2_13](#) (-4.0)

Q2_29_MAX Macro

File

libq.h

C

```
#define _Q2_29_MAX ((_Q2_29)0x7FFFFFFF) // Maximum value of _Q2_29 (~4.0)
```

Description

Maximum value of [_Q2_29](#) (~4.0)

Q2_29_MIN Macro

File

libq.h

C

```
#define _Q2_29_MIN ((_Q2_29)0x80000000) // Minimum value of _Q2_29 (-4.0)
```

Description

Minimum value of [_Q2_29](#) (-4.0)

Q3_12_MAX Macro

File

libq.h

C

```
#define _Q3_12_MAX ((_Q3_12)0x7FFF) // Maximum value of _Q3_12 (~8.0)
```

Description

Maximum value of [_Q3_12](#) (~8.0)

Q3_12_MIN Macro

File

libq.h

C

```
#define _Q3_12_MIN ((_Q3_12)0x8000) // Minimum value of _Q3_12 (-8.0)
```

Description

Minimum value of [_Q3_12](#) (-8.0)

Q31_MAX Macro

File

libq.h

C

```
#define _Q31_MAX ((_Q31)0x7FFFFFFF) // Maximum value of _Q31 (~1.0)
```

Description

Maximum value of [_Q31](#) (~1.0)

Q31_MIN Macro

File

libq.h

C

```
#define _Q31_MIN ((_Q31)0x80000000) // Minimum value of _Q31 (-1.0)
```

Description

Minimum value of [_Q31](#) (-1.0)

Q4_11_MAX Macro

File

libq.h

C

```
#define _Q4_11_MAX ((_Q4_11)0x7FFF) // Maximum value of _Q4_11 (~16.0)
```

Description

Maximum value of [_Q4_11](#) (~16.0)

Q4_11_MIN Macro

File

libq.h

C

```
#define _Q4_11_MIN ((_Q4_11)0x8000) // Minimum value of _Q4_11 (-16.0)
```

Description

Minimum value of [_Q4_11](#) (-16.0)

Q5_10_MAX Macro

File

libq.h

C

```
#define _Q5_10_MAX ((_Q5_10)0x7FFF) // Maximum value of _Q5_10 (~32.0)
```

Description

Maximum value of [_Q5_10](#) (~32.0)

Q5_10_MIN Macro

File

libq.h

C

```
#define _Q5_10_MIN ((_Q5_10)0x8000) // Minimum value of _Q5_10 (-32.0)
```

Description

Minimum value of [_Q5_10](#) (-32.0)

Q7_8_MAX Macro

File

libq.h

C

```
#define _Q7_8_MAX ((_Q7_8)0x7FFF) // Maximum value of _Q7_8 (~128.0)
```

Description

Maximum value of [_Q7_8](#) (~128.0)

Q7_8_MIN Macro

File

libq.h

C

```
#define _Q7_8_MIN ((_Q7_8)0x8000) // Minimum value of _Q7_8 (-128.0)
```

Description

Minimum value of [_Q7_8](#) (-128.0)

Q0_15 Type

File

libq.h

C

```
typedef int16_t _Q0_15;
```

Description

1 sign bit, 15 bits right of radix

_Q0_31 Type

File

[libq.h](#)

C

```
typedef int32_t _Q0_31;
```

Description

1 sign bit, 31 bits right of radix

_Q15 Type

File

[libq.h](#)

C

```
typedef _Q0_15 _Q15;
```

Description

Short name for [_Q0_15](#)

_Q15_16 Type

File

[libq.h](#)

C

```
typedef int32_t _Q15_16;
```

Description

1 sign bit, 15 bits left of radix, 16 bits right of radix

_Q16 Type

File

[libq.h](#)

C

```
typedef _Q15_16 _Q16;
```

Description

Short name for [_Q15_16](#)

_Q2_13 Type

File

[libq.h](#)

C

```
typedef int16_t _Q2_13;
```

Description

1 sign bit, 2 bits left of radix, 13 bits right of radix

Q2_29 Type

File

[libq.h](#)

C

```
typedef int32_t _Q2_29;
```

Description

1 sign bit, 2 bits left of radix, 29 bits right of radix

Q3_12 Type

File

[libq.h](#)

C

```
typedef int16_t _Q3_12;
```

Description

1 sign bit, 3 bits left of radix, 12 bits right of radix

Q31 Type

File

[libq.h](#)

C

```
typedef _Q0_31 _Q31;
```

Description

Short name for `_Q_0_31`

Q4_11 Type

File

[libq.h](#)

C

```
typedef int16_t _Q4_11;
```

Description

1 sign bit, 4 bits left of radix, 11 bits right of radix

Q5_10 Type

File

[libq.h](#)

C

```
typedef int16_t _Q5_10;
```

Description

1 sign bit, 5 bits left of radix, 10 bits right of radix

_Q7_8 Type

File

[libq.h](#)

C

```
typedef int16_t _Q7_8;
```

Description

1 sign bit, 7 bits left of radix, 8 bits right of radix

LIBQ_H Macro

File

[libq.h](#)

C

```
#define _LIBQ_H
```

Description

Guards against multiple inclusion

Files

Files

Name	Description
libq.h	Optimized fixed point math functions for the PIC32MZ families of devices with microAptiv core features.


















Description















This section lists the source and header files used by the LibQ Fixed-Point Math Library.

libq.h

Optimized fixed point math functions for the PIC32MZ families of devices with microAptiv core features.

Functions

	Name	Description
	_LIBQ_Q15_cos_Q2_13	Calculates the value of cosine(x).
	_LIBQ_Q15_sin_Q2_13	Calculates the value of sine(x).
	_LIBQ_Q15FromFloat	Converts a float to a _Q15 value.
	_LIBQ_Q15FromString	ASCII to _Q15 conversion.
	_LIBQ_Q15Rand	Generate a _Q15 random number.
	_LIBQ_Q16_tan_Q2_29	Calculates the value of tan(x).
	_LIBQ_Q16Div	_Q16 fixed point divide.
	_LIBQ_Q16Exp	Calculates the exponential function e ^x .
	_LIBQ_Q16Power	Calculates the value of x raised to the y power (x ^y).
	_LIBQ_Q16Sqrt	Square root of a positive _Q16 fixed point value.
	_LIBQ_Q2_13_acos_Q15	Calculates the value of acos(x).
	_LIBQ_Q2_13_asin_Q15	Calculates the asin value of asin(x).
	_LIBQ_Q2_13_atan_Q7_8	Calculates the value of atan(x).
	_LIBQ_Q2_13_atan2_Q7_8	Calculates the value of atan2(y, x).
	_LIBQ_Q2_29_acos_Q31	Calculates the value of acos(x).
	_LIBQ_Q2_29_acos_Q31_Fast	Calculates the value of acos(x). This function executes faster than _LIBQ_Q2_29_acos_Q31 but is less precise.
	_LIBQ_Q2_29_asin_Q31	Calculates the value of asin(x).

	_LIBQ_Q2_29_asin_Q31_Fast	Calculates the value of $\text{asin}(x)$. This function executes faster than the _LIBQ_Q2_29_asin_Q31 function, but is less precise.
	_LIBQ_Q2_29_atan_Q16	Calculates the value of $\text{atan}(x)$.
	_LIBQ_Q2_29_atan2_Q16	Calculates the value of $\text{atan2}(y, x)$.
	_LIBQ_Q3_12_log10_Q16	Calculates the value of $\text{Log}_{10}(x)$.
	_LIBQ_Q31_cos_Q2_29	Calculates the value of $\text{cosine}(x)$.
	_LIBQ_Q31_sin_Q2_29	Calculates the value of $\text{sine}(x)$.
	_LIBQ_Q31FromFloat	Converts a float to a _Q31 value.
	_LIBQ_Q31Rand	Generate a _Q31 random number.
	_LIBQ_Q4_11_ln_Q16	Calculates the natural logarithm $\ln(x)$.
	_LIBQ_Q5_10_log2_Q16	Calculates the value of $\log_2(x)$.
	_LIBQ_Q7_8_tan_Q2_13	Calculates the value of $\tan(x)$.
	_LIBQ_ToFloatQ15	Converts a _Q15 value to a float.
	_LIBQ_ToFloatQ31	Converts a _Q31 value to a float.
	_LIBQ_ToStringQ15	_Q15 to ASCII conversion.

Macros

Name	Description
_LIBQ_H	Guards against multiple inclusion
_Q15_MAX	Maximum value of _Q15 (~1.0)
_Q15_MIN	Minimum value of _Q15 (-1.0)
_Q16_MAX	Maximum value of _Q16 (~32768.0)
_Q16_MIN	Minimum value of _Q16 (-32768.0)
_Q2_13_MAX	Maximum value of _Q2_13 (~4.0)
_Q2_13_MIN	Minimum value of _Q2_13 (-4.0)
_Q2_29_MAX	Maximum value of _Q2_29 (~4.0)
_Q2_29_MIN	Minimum value of _Q2_29 (-4.0)
_Q3_12_MAX	Maximum value of _Q3_12 (~8.0)
_Q3_12_MIN	Minimum value of _Q3_12 (-8.0)
_Q31_MAX	Maximum value of _Q31 (~1.0)
_Q31_MIN	Minimum value of _Q31 (-1.0)
_Q4_11_MAX	Maximum value of _Q4_11 (~16.0)
_Q4_11_MIN	Minimum value of _Q4_11 (-16.0)
_Q5_10_MAX	Maximum value of _Q5_10 (~32.0)
_Q5_10_MIN	Minimum value of _Q5_10 (-32.0)
_Q7_8_MAX	Maximum value of _Q7_8 (~128.0)
_Q7_8_MIN	Minimum value of _Q7_8 (-128.0)

Types

Name	Description
_Q0_15	1 sign bit, 15 bits right of radix
_Q0_31	1 sign bit, 31 bits right of radix
_Q15	Short name for _Q0_15
_Q15_16	1 sign bit, 15 bits left of radix, 16 bits right of radix
_Q16	Short name for _Q15_16
_Q2_13	1 sign bit, 2 bits left of radix, 13 bits right of radix
_Q2_29	1 sign bit, 2 bits left of radix, 29 bits right of radix
_Q3_12	1 sign bit, 3 bits left of radix, 12 bits right of radix
_Q31	Short name for _Q0_31
_Q4_11	1 sign bit, 4 bits left of radix, 11 bits right of radix
_Q5_10	1 sign bit, 5 bits left of radix, 10 bits right of radix
_Q7_8	1 sign bit, 7 bits left of radix, 8 bits right of radix

Description

The LibQ Fixed-Point Math Library provides fixed-point math functions that are optimized for performance on the PIC32MZ families of devices that have microAptiv core features. All functions are optimized for speed. This header file specifies characteristics of each function, including execution

time, memory size, and resolution.

Signed fixed point types are defined as follows:

Q_n_m where:

- n is the number of data bits to the left of the radix point
- m is the number of data bits to the right of the radix point
- a signed bit is implied

For convenience, short names are also defined:

Exact Name (& Bits) Required Short Name [_Q0_15](#) (16) [_Q15](#); [_Q15_16](#) (32) [_Q16](#); [_Q0_31](#) (32) [_Q31](#)

Functions in the library are prefixed with the type of the return value. For example, [_LIBQ_Q16Sqrt](#) returns a Q16 value equal to the square root of its argument.

Argument types do not always match the return type. Refer to the function prototype for a specification of its arguments.

In cases where the return value is not a fixed point type, the argument type is appended to the function name. For example, [_LIBQ_ToFloatQ31](#) accepts a type [_Q31](#) argument.

In some cases, both the return type and the argument type are specified within the function name. For example,

Function Name (Return Type) [Argument Type]: [_LIBQ_Q15_sin_Q2_13](#) ([_Q15](#)) [[_Q2_13](#)]; [_LIBQ_Q31_sin_Q2_29](#) ([_Q31](#)) [[_Q2_29](#)]

Table of LIBQ functions:

Divide: [_Q16_LIBQ_Q16Div](#) ([_Q16](#) dividend, [_Q16](#) divisor);

Square root: [_Q16_LIBQ_Q16Sqrt](#) ([_Q16](#) x);

Exponential: [_Q16_LIBQ_Q16Exp](#) ([_Q16](#) x);

Log: [_Q4_11_LIBQ_Q4_11_ln_Q16](#) ([_Q16](#) x); [_Q3_12_LIBQ_Q3_12_log10_Q16](#) ([_Q16](#) x); [_Q5_10_LIBQ_Q5_10_log2_Q16](#) ([_Q16](#) x);

Power: [_Q16_LIBQ_Q16Power](#) ([_Q16](#) x, [_Q16](#) y);

Sine: [_Q15_LIBQ_Q15_sin_Q2_13](#) ([_Q2_13](#) x); [_Q31_LIBQ_Q31_sin_Q2_29](#) ([_Q2_29](#) x);

Cosine: [_Q15_LIBQ_Q15_cos_Q2_13](#) ([_Q2_13](#) x); [_Q31_LIBQ_Q31_cos_Q2_29](#) ([_Q2_29](#) x);

Tangent: [_Q7_8_LIBQ_Q7_8_tan_Q2_13](#) ([_Q2_13](#) x); [_Q16z_LIBQ_Q16_tan_Q2_29](#) ([_Q2_29](#) x);

Arcsin: [_Q2_13_LIBQ_Q2_13_asin_Q15](#) ([_Q15](#) x); [_Q2_29_LIBQ_Q2_29_asin_Q31](#) ([_Q31](#) x); [_Q2_29_LIBQ_Q2_29_asin_Q31_Fast](#) ([_Q31](#) x);

Arccos: [_Q2_13_LIBQ_Q2_13_acos_Q15](#) ([_Q15](#) x); [_Q2_29_LIBQ_Q2_29_acos_Q31](#) ([_Q31](#) x); [_Q2_29_LIBQ_Q2_29_acos_Q31_Fast](#) ([_Q31](#) x);

Arctan: [_Q2_13_LIBQ_Q2_13_atan_Q7_8](#) ([_Q7_8](#) x); [_Q2_29_LIBQ_Q2_29_atan_Q16](#) ([_Q16](#) x);

Arctan2: [_Q2_13_LIBQ_Q2_13_atan2_Q7_8](#) ([_Q7_8](#) y, [_Q7_8](#) x); [_Q2_29_LIBQ_Q2_29_atan2_Q16](#) ([_Q16](#) y, [_Q16](#) x);

Random number: [_Q15_LIBQ_Q15Rand](#) (int64_t &pSeed); [_Q31_LIBQ_Q31Rand](#) (int64_t &pSeed);

Float: float [_LIBQ_ToFloatQ31](#) ([_Q31](#) x); float [_LIBQ_ToFloatQ15](#) ([_Q15](#) x); [_Q31_LIBQ_Q31FromFloat](#) (float x); [_Q15_LIBQ_Q15FromFloat](#) (float x);

String: void [_LIBQ_ToStringQ15](#) ([_Q15](#) x, char &s); [_Q15_LIBQ_Q15FromString](#) (char &s);

File Name

libq.h

Company

Microchip Technology Inc.

Index

- - `_LIBQ_C_H_` macro 113
 - `_LIBQ_H` macro 157
 - `_LIBQ_Q15_cos_Q2_13` function 136
 - `_LIBQ_Q15_sin_Q2_13` function 135
 - `_LIBQ_Q15FromFloat` function 147
 - `_LIBQ_Q15FromString` function 149
 - `_LIBQ_Q15Rand` function 145
 - `_LIBQ_Q16_tan_Q2_29` function 138
 - `_LIBQ_Q16Div` function 131
 - `_LIBQ_Q16Exp` function 135
 - `_LIBQ_Q16Power` function 134
 - `_LIBQ_Q16Sqrt` function 131
 - `_LIBQ_Q2_13_acos_Q15` function 141
 - `_LIBQ_Q2_13_asin_Q15` function 139
 - `_LIBQ_Q2_13_atan_Q7_8` function 143
 - `_LIBQ_Q2_13_atan2_Q7_8` function 144
 - `_LIBQ_Q2_29_acos_Q31` function 141
 - `_LIBQ_Q2_29_acos_Q31_Fast` function 142
 - `_LIBQ_Q2_29_asin_Q31` function 139
 - `_LIBQ_Q2_29_asin_Q31_Fast` function 140
 - `_LIBQ_Q2_29_atan_Q16` function 143
 - `_LIBQ_Q2_29_atan2_Q16` function 145
 - `_LIBQ_Q3_12_log10_Q16` function 132
 - `_LIBQ_Q31_cos_Q2_29` function 137
 - `_LIBQ_Q31_sin_Q2_29` function 136
 - `_LIBQ_Q31FromFloat` function 147
 - `_LIBQ_Q31Rand` function 146
 - `_LIBQ_Q4_11_In_Q16` function 133
 - `_LIBQ_Q5_10_log2_Q16` function 133
 - `_LIBQ_Q7_8_tan_Q2_13` function 138
 - `_LIBQ_ToFloatQ15` function 148
 - `_LIBQ_ToFloatQ31` function 149
 - `_LIBQ_ToStringQ15` function 150
 - `_PARAM_EQUAL_FILTER` structure 89
 - `_PARAM_EQUAL_FILTER_16` structure 90
 - `_PARAM_EQUAL_FILTER_32` structure 90
 - `_Q0_15` type 154
 - `_Q0_31` type 155
 - `_Q15` type 155
 - `_Q15_16` type 155
 - `_Q15_MAX` macro 151
 - `_Q15_MIN` macro 151
 - `_Q16` type 155
 - `_Q16_MAX` macro 151
 - `_Q16_MIN` macro 151
 - `_Q2_13` type 155
 - `_Q2_13_MAX` macro 152
 - `_Q2_13_MIN` macro 152
 - `_Q2_29` type 156
 - `_Q2_29_MAX` macro 152
 - `_Q2_29_MIN` macro 152
 - `_Q3_12` type 156
 - `_Q3_12_MAX` macro 152
 - `_Q3_12_MIN` macro 153
 - `_Q31` type 156
 - `_Q31_MAX` macro 153
 - `_Q31_MIN` macro 153
 - `_Q4_11` type 156
 - `_Q4_11_MAX` macro 153
 - `_Q4_11_MIN` macro 153
 - `_Q5_10` type 156
 - `_Q5_10_MAX` macro 154
 - `_Q5_10_MIN` macro 154
 - `_Q7_8` type 157
 - `_Q7_8_MAX` macro 154
 - `_Q7_8_MIN` macro 154
- ## B
- biquad16 structure 88
 - `BITMASKFRACT16` macro 113
 - `BITMASKFRACT32` macro 113
- ## D
- DSP Fixed-Point Math Library 3
 - `dsp.h` 92
 - `DSP_ComplexAdd32` function 8
 - `DSP_ComplexConj16` function 9
 - `DSP_ComplexConj32` function 9
 - `DSP_ComplexDotProd32` function 10
 - `DSP_ComplexMult32` function 11
 - `DSP_ComplexScalarMult32` function 12
 - `DSP_ComplexSub32` function 12
 - `DSP_FilterFIR32` function 13
 - `DSP_FilterFIRDecim32` function 14
 - `DSP_FilterFIRInterp32` function 15
 - `DSP_FilterIIR16` function 16
 - `DSP_FilterIIRBQ16` function 17
 - `DSP_FilterIIRBQ16_cascade8` function 18
 - `DSP_FilterIIRBQ16_cascade8_fast` function 20
 - `DSP_FilterIIRBQ16_fast` function 22
 - `DSP_FilterIIRBQ16_parallel8` function 23
 - `DSP_FilterIIRBQ16_parallel8_fast` function 24
 - `DSP_FilterIIRBQ32` function 26
 - `DSP_FilterIIRSetup16` function 27
 - `DSP_FilterLMS16` function 27
 - `DSP_MatrixAdd32` function 29
 - `DSP_MatrixEqual32` function 30
 - `DSP_MatrixInit32` function 30
 - `DSP_MatrixMul32` function 31
 - `DSP_MatrixScale32` function 32
 - `DSP_MatrixSub32` function 33
 - `DSP_MatrixTranspose32` function 34
 - `DSP_TransformFFT16` function 35
 - `DSP_TransformFFT16_setup` function 36
 - `DSP_TransformFFT32` function 37
 - `DSP_TransformFFT32_setup` function 37
 - `DSP_TransformIFFT16` function 38
 - `DSP_TransformWindow_Bart16` function 39
 - `DSP_TransformWindow_Bart32` function 40
 - `DSP_TransformWindow_Black16` function 41
 - `DSP_TransformWindow_Black32` function 41
 - `DSP_TransformWindow_Cosine16` function 42

DSP_TransformWindow_Cosine32 function 43
 DSP_TransformWindow_Hamm16 function 44
 DSP_TransformWindow_Hamm32 function 44
 DSP_TransformWindow_Hann16 function 45
 DSP_TransformWindow_Hann32 function 46
 DSP_TransformWindow_Kaiser16 function 47
 DSP_TransformWindow_Kaiser32 function 47
 DSP_TransformWinInit_Bart16 function 48
 DSP_TransformWinInit_Bart32 function 49
 DSP_TransformWinInit_Black16 function 49
 DSP_TransformWinInit_Black32 function 50
 DSP_TransformWinInit_Cosine16 function 51
 DSP_TransformWinInit_Cosine32 function 51
 DSP_TransformWinInit_Hamm16 function 52
 DSP_TransformWinInit_Hamm32 function 52
 DSP_TransformWinInit_Hann16 function 53
 DSP_TransformWinInit_Hann32 function 54
 DSP_TransformWinInit_Kaiser16 function 54
 DSP_TransformWinInit_Kaiser32 function 55
 DSP_VectorAbs16 function 55
 DSP_VectorAbs32 function 56
 DSP_VectorAdd16 function 57
 DSP_VectorAdd32 function 58
 DSP_VectorAddc16 function 58
 DSP_VectorAddc32 function 59
 DSP_VectorAutocorr16 function 60
 DSP_VectorBexp16 function 61
 DSP_VectorBexp32 function 61
 DSP_VectorChkEqu32 function 62
 DSP_VectorCopy function 63
 DSP_VectorCopyReverse32 function 64
 DSP_VectorDivC function 64
 DSP_VectorDotp16 function 65
 DSP_VectorDotp32 function 66
 DSP_VectorExp function 67
 DSP_VectorFill function 67
 DSP_VectorLn function 68
 DSP_VectorLog10 function 69
 DSP_VectorLog2 function 70
 DSP_VectorMax32 function 70
 DSP_VectorMaxIndex32 function 71
 DSP_VectorMean32 function 72
 DSP_VectorMin32 function 72
 DSP_VectorMinIndex32 function 73
 DSP_VectorMul16 function 74
 DSP_VectorMul32 function 75
 DSP_VectorMulc16 function 75
 DSP_VectorMulc32 function 76
 DSP_VectorNegate function 77
 DSP_VectorRecip function 78
 DSP_VectorRMS16 function 79
 DSP_VectorShift function 79
 DSP_VectorSqrt function 80
 DSP_VectorStdDev16 function 81
 DSP_VectorSub16 function 82
 DSP_VectorSub32 function 82
 DSP_VectorSumSquares16 function 83
 DSP_VectorSumSquares32 function 84

DSP_VectorVari16 function 84
 DSP_VectorVariance function 85
 DSP_VectorZeroPad function 86

E

Exponent16ToQFloat32 macro 122

F

Files 92, 123, 157

- DSP Fixed-Point Math Library 92
- LibQ Fixed-Point 'C' Math Library 123
- LibQ Fixed-Point Math Library 157

FI2Fract16 macro 113

FI2Fract32 macro 114

FI2Fxpnt macro 114

FI2Fxpnt16 macro 114

FI2Fxpnt32 macro 115

FI2Int16 macro 115

FI2Int32 macro 116

FI2QFloat32 macro 123

FrMax macro 116

FrMin macro 116

Fx16Norm function 108

Fx32Norm function 108

FxQFloat32 variable 122

I

i16 type 123

int16c structure 88

int32c structure 89

Introduction 3, 97, 126

L

LibQ Fixed-Point 'C' Math Library 97

LibQ Fixed-Point Math Library 126

libq.h 157

libq_C.h 124

libq_q15_Abs_q15 function 100

libq_q15_Add_q15_q15 function 100

libq_q15_DivisionWithSaturation_q15_q15 function 100

libq_q15_ExpAvg_q15_q15_q1d15 function 106

libq_q15_ExtractH_q31 function 101

libq_q15_ExtractL_q31 function 101

libq_q15_MacR_q31_q15_q15 function 102

libq_q15_MsuR_q31_q15_q15 function 102

libq_q15_Negate_q15 function 101

libq_q15_RoundL_q31 function 102

libq_q15_ShiftLeft_q15_i16 function 109

libq_q15_ShiftRight_q15_i16 function 109

libq_q15_ShiftRightRound_q15_i16 function 110

libq_q15_Sub_q15_q15 function 104

libq_q1d15_Sin_q10d6 function 103

libq_q20d12_Sin_q20d12 function 107

libq_q31_Abs_q31 function 103

libq_q31_Add_q31_q31 function 104

libq_q31_DepositH_q15 function 104

libq_q31_DepositL_q15 function 105

libq_q31_Mac_q31_q15_q15 function 106

libq_q31_Msu_q31_q15_q15 function 107

libq_q31_Mult2_q15_q15 function 110
 libq_q31_Multi_q15_q31 function 105
 libq_q31_Negate_q31 function 106
 libq_q31_ShiftLeft_q31_i16 function 107
 libq_q31_ShiftRight_q31_i16 function 111
 libq_q31_ShiftRightRound_q31_i16 function 111
 libq_q31_Sub_q31_q31 function 112
 Library Interface 5, 98, 129
 DSP Fixed-Point Math Library 5
 LibQ Fixed-Point 'C' Math Library 98
 LibQ Fixed-Point Math Library 129
 Library Overview 3, 127
 DSP-Fixed Point Math Library 3
 LibQ Fixed-Point Math Library 127
 LOG102Q5D11 macro 116

M

Math Libraries Help 3
 matrix32 structure 89
 MAX16 macro 91
 MAX32 macro 91
 MAXFRACT16 macro 117
 MAXFRACT32 macro 117
 MAXINT16 macro 117
 MAXINT32 macro 117
 MAXPFLOAT32 macro 117
 MIN16 macro 91
 MIN32 macro 91
 MINFRACT16 macro 118
 MINFRACT32 macro 118
 MININT16 macro 118
 MININT32 macro 118
 MINPFLOAT32 macro 118
 MSBBITFRACT16 macro 119
 MSBBITFRACT32 macro 119
 mul16 function 87
 mul16r function 87
 mul32 function 87

N

NINETYQ10D22 macro 119
 NINETYQ10D6 macro 119
 NORMNEGFRACT16 macro 119
 NORMNEGFRACT32 macro 120
 NORMPOSFRACT16 macro 120
 NORMPOSFRACT32 macro 120
 NUMBITSFRACT16 macro 120
 NUMBITSFRACT32 macro 120

O

ONEEIGHTYQ10D22 macro 121
 ONEEIGHTYQ10D6 macro 121

P

PARAM_EQUAL_FILTER structure 89
 PARAM_EQUAL_FILTER_16 structure 90
 PARAM_EQUAL_FILTER_32 structure 90
 PARAM_FILTER_GAIN structure 91

Q

q15 type 112
 q31 type 112
 q63 type 112

R

ROUNDFRACT32 macro 121

S

SAT16 function 87
 SAT16N function 87
 SAT16P function 88

T

Table of Library Functions 4, 128
 THREESIXTYQ10D22 macro 121
 THREESIXTYQ10D6 macro 121
 TWOSEVENTYQ10D22 macro 122
 TWOSEVENTYQ10D6 macro 122

U

UNITYFLOAT macro 122
 Using the Library 3, 98, 127
 DSP Fixed-Point Math Library 3
 LibQ Fixed-Point 'C' Math Library 98
 LibQ Fixed-Point Math Library 127

V

Volume V: MPLAB Harmony Framework Reference 2