XMath - Programming Guide

XMOS

Table of Contents

1 Introduction

`lib_xcore_math` is a library of optimized math functions for taking advantage of the vector processing unit (VPU) on the XMOS XS3 architecture. Included in the library are functions for block floating-point arithmetic, fast Fourier transforms, linear algebra, discrete cosine transforms, linear filtering and more.

See *Getting Started* to get going.

1.1   On GitHub

Find the latest version of `lib_xcore_math` at https://github.com/xmos/lib_xcore_math.

2 Getting Started

2.1 Overview

`lib_xcore_math` is a library containing efficient implementations of various mathematical operations that may be required in an embedded application. In particular, this library is geared towards operations which work on vectors or arrays of data, including vectorized arithmetic, linear filtering, and fast Fourier transforms.

This library comprises several sub-APIs. Grouping of operations into sub-APIs is a matter of conceptual convenience. In general, functions from a given API share a common prefix indicating which API the function comes from, or the type of object on which it acts. Additionally, there is some interdependence between these APIs.

These APIs are:

- *Block floating-point (BFP) API* – High-level API providing operations on BFP vectors. See *Block Floating-Point Background* for an introduction to block floating-point. These functions manage the exponents and headroom of input and output BFP vectors to avoid overflow and underflow conditions.

- *Vector/Array API* – Lower-level API which is used heavily by the BFP API. As such, the operations available in this API are similar to those in the BFP API, but the user will have to manage exponents and headroom on their own. Many of these routines are implemented directly in optimized assembly to use the hardware as efficiently as possible.

- *Scalar API* – Provides various operations on scalar objects. In particular, these operations focus on simple arithmetic operations applied to non-IEEE 754 floating-point objects, as well as optimized operations which are applied to IEEE 754 `floats`.

- *Filtering API* – Provides access to linear filtering operations, including 16- and 32-bit FIR filters and 32-bit biquad filters.

- *Fast Fourier Transform (FFT) API* – Provides both low-level and block floating-point FFT implementations. Optimized FFT implementations are provided for real signals, pairs of real signals, and for complex signals.

- *Discrete Cosine Transform (DCT) API* – Provides functions which implement the type-II ('forward') and type-III ('inverse') DCT for a variety of block lengths. Also provides a fast 8x8 two dimensional forward and inverse DCT.

All APIs are accessed by including the single header file:

```
#include "xmath/xmath.h"
```

2.2 Building

This library makes use of the CMake build system. See the GitHub page for instructions on obtaining this library's source code and including it in your application's build.

2.3 Usage

The following sections are intended to give the reader a general sense of how to use the API.

2.3.1 BFP API

In the BFP API the BFP vectors are C structures such as `bfp_s16_t`, `bfp_s32_t`, or `bfp_complex_s32_t`, backed by a memory buffer. These objects contain a pointer to the data carrying the content (mantissas) of the vector, as well as information about the length, headroom and exponent of the BFP vector.

Below is the definition of *bfp_s32_t* from xmath/types.h.

```
C_TYPE
typedef struct {
    /** Pointer to the underlying element buffer.*/
    int32_t* data;
    /** Exponent associated with the vector. */
    exponent_t exp;
    /** Current headroom in the ``data[]`` */
    headroom_t hr;
    /** Current size of ``data[]``, expressed in elements */
    unsigned length;
    /** BFP vector flags. Users should not normally modify these manually. */
    bfp_flags_e flags;
} bfp_s32_t;
```

The *32-bit BFP functions* take *bfp_s32_t* pointers as input and output parameters.

Functions in the BFP API generally are prefixed with `bfp_`. More specifically, functions where the 'main' operands are 32-bit BFP vectors are prefixed with `bfp_s32_`, whereas functions where the 'main' operands are complex 16-bit BFP vectors are prefixed with `bfp_complex_s16_`, and so on for the other BFP vector types.

2.3.1.1 Initializing BFP Vectors

Before calling these functions, the BFP vectors represented by the arguments must be initialized. For *bfp_s32_t* this is accomplished with *bfp_s32_init()*. Initialization requires that a buffer of sufficient size be provided to store the mantissa vector, as well as an initial exponent. If the first usage of a BFP vector is as an output, then the exponent will not matter, but the object must still be initialized before use. Additionally, the headroom of the vector may be computed upon initialization; otherwise it is set to 0.

Here is an example of a 32-bit BFP vector being initialized.

```
#define LEN (20)

//The object representing the BFP vector
bfp_s32_t bfp_vect;

// buffer backing bfp_vect
int32_t data_buffer[LEN];
for(int i = 0; i < LEN; i++) data_buffer[i] = i;

// The initial exponent associated with bfp_vect
exponent_t initial_exponent = 0;

// If non-zero, `bfp_s32_init()` will compute headroom currently present in data_buffer.
// Otherwise, headroom is initialized to 0 (which is always safe but may not be optimal)
unsigned calculate_headroom = 1;
```

(continues on next page)

```
// Initialize the vector object
bfp_s32_init(&bfp_vec, data_buffer, initial_exponent, LEN, calculate_headroom);

// Go do stuff with bfp_vect
...
```

Once initialized, the exponent and mantissas of the vector can be accessed by `bfp_vect.exp` and `bfp_vect.data[]` respectively, with the logical (floating-point) value of element `k` being given by $bfp\_vect.data[k] \cdot 2^{bfp\_vect.exp}$.

2.3.1.2 BFP Arithmetic Functions

The following snippet shows a function `foo()` which takes 3 BFP vectors, `a`, `b` and `c`, as arguments. It multiplies together `a` and `b` element-wise, and then subtracts `c` from the product. In this example both operations are performed in-place on `a`. (See *bfp_s32_mul()* and *bfp_s32_sub()* for more information about those functions)

```
void foo(bfp_s32_t* a, const bfp_s32_t* b, const bfp_s32_t* c)
{
    // Multiply together a and b, updating a with the result.
    bfp_s32_mul(a, a, b);

    // Subtract c from the product, again updating a with the result.
    bfp_s32_sub(a, a, c);
}
```

The caller of `foo()` can then access the results through `a`. Note that the pointer `a->data` was not modified during this call.

2.3.2 Vector API

The functions in the lower-level vector API are optimized for performance. They do very little to protect the user from mangling their data by arithmetic saturation/overflows or underflows (although they do provide the means to prevent this).

Functions in the vector API are generally prefixed with `vect_`. For example, functions which operate primarily on 16-bit vectors are prefixed with `vect_s16_`.

Some functions are prefixed with `chunk_` instead of `vect_`. A "chunk" is just a vector with a fixed memory footprint (currently 32 bytes, or 8 32-bit elements) meant to match the width of the architecture's vector registers.

As an example of a function from the vector API, see *vect_s32_mul()* (from `vect_s32.h`)), which multiplies together two `int32_t` vectors element by element.

```
C_API
headroom_t vect_s32_mul(
    int32_t a[],
    const int32_t b[],
    const int32_t c[],
    const unsigned length,
    const right_shift_t b_shr,
    const right_shift_t c_shr);
```

This function takes two `int32_t` arrays, `b` and `c`, as inputs and one `int32_t` array, `a`, as output (in the case of *vect_s32_mul()*, it is safe to have `a` point to the same buffer as `b` or `c`, computing the result in-place). `length` indicates the number of elements in each array. The final two parameters, `b_shr` and `c_shr`, are the arithmetic right-shifts applied to each element of `b` and `c` before they are multiplied together.

Why the right-shifts? In the case of 32-bit multiplication, the largest possible product is 2^{62}, which will not fit in the 32-bit output vector. Applying positive arithmetic right-shifts to the input vectors reduces the largest possible product. So, the shifts are there to manage the headroom/size of the resulting product in order to maximize precision while avoiding overflow or saturation.

Contrast this with *vect_s16_mul()*:

```
C_API
headroom_t vect_s16_mul(
    int16_t a[],
    const int16_t b[],
    const int16_t c[],
    const unsigned length,
    const right_shift_t a_shr);
```

The parameters are similar here, but instead of `b_shr` and `c_shr`, there's only an `a_shr`. In this case, the arithmetic right-shift `a_shr` is applied to the *products* of `b` and `c`. In this case the right-shift is also *unsigned* – it can only be used to reduce the size of the product.

Shifts like those in these two examples are very common in the vector API, as they are the main mechanism for managing exponents and headroom. Whether the shifts are applied to inputs, outputs, both, or only one input will depend on a number of factors. In the case of *vect_s32_mul()* they are applied to inputs because the XS3 VPU includes a compulsory (hardware) right-shift of 30 bits on all products of 32-bit numbers, and so often inputs may need to be *left*-shifted (negative shift) in order to avoid underflows. In the case of *vect_s16_mul()*, this is unnecessary because no compulsory shift is included in 16-bit multiply-accumulates.

Both *vect_s32_mul()* and *vect_s16_mul()* return the headroom of the output vector `a`.

Functions in the vector API are in many cases closely tied to the instruction set architecture for XS3. As such, if more efficient algorithms are found to perform an operation these low-level API functions are more likely to change in future versions.

3 Block Floating-Point Background

3.1 Block Floating-Point Vectors

A standard (IEEE) floating-point object can exist either as a scalar, e.g.

```
//Single IEEE floating-point variable
float foo;
```

or as a vector, e.g.

```
//Array of IEEE floating-point variables
float foo[20];
```

Standard floating-point values carry both a mantissa m and an exponent p, such that the logical value represented by such a variable is $m \cdot 2^p$. When you have a vector of standard floating-point values, each element of the vector carries its own mantissa and its own exponent: $m[k] \cdot 2^{p[k]}$.


```
float foo[20];
```
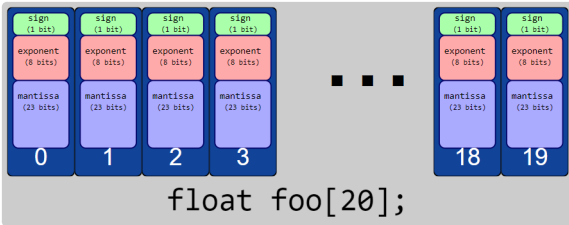
By contrast, block floating-point objects have a vector of mantissas $\bar{m}$ which all share the same exponent p, such that the logical value of the element at index k is $m[k] \cdot 2^p$.

```
struct {
    // Array of mantissas
    int32_t mant[20];
    // Shared exponent
    int32_t exp;
} bfp_vect;
```



```
bfp_vect.mant[]                    bfp_vect.exp
```
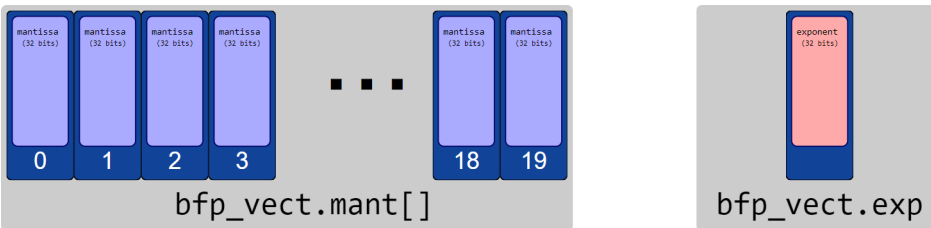
3.2 Headroom

With a given exponent, p, the largest value that can be represented by a 32-bit BFP vector is given by a maximal mantissa $(2^{31}-1)$, for a logical value of $(2^{31}-1)\cdot 2^p$. The smallest non-zero value that an element can represent is $1 \cdot 2^p$.

Because all elements must share a single exponent, in order to avoid overflow or saturation of the largest magnitude values, the exponent of a BFP vector is constrained by the element with the largest (logical) value. The drawback to this is that when the elements of a BFP vector represent a large dynamic range – that is, where the largest magnitude element is many, many times larger than the smallest (non-zero) magnitude element – the smaller magnitude elements effectively have fewer bits of precision.

Consider a 2-element BFP vector intended to carry the values 2^{20} and $255 \cdot 2^{-10}$. One way this vector can be represented is to use an exponent of 0.

```
struct {
    int32_t mant[2];
    int32_t exp;
} vect = { { (1<<20), (0xFF >> 10) }, 0 };
```
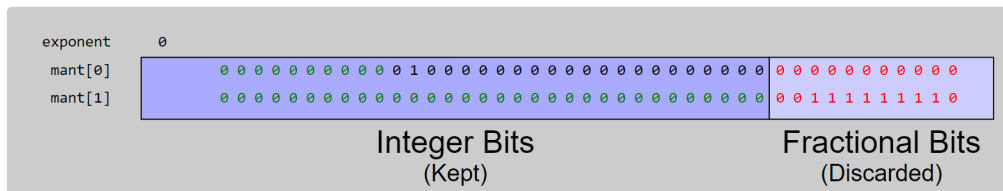

In the diagram above, the fractional bits (shown in red text) are discarded, as the mantissa is only 32 bits. Then, with 0 as the exponent, `mant[1]` underflows to 0. Meanwhile, the 12 most significant bits of `mant[0]` are all zeros.

The headroom of a signed integer is the number of *redundant* leading sign bits. Equivalently, it is the number of bits that a mantissa can be left-shifted without losing any information. In the the diagram, the bits corresponding to headroom are shown in green text. Here `mant[0]` has 10 bits of headroom and `mant[1]` has a full 32 bits of headroom. (`mant[0]` does not have 11 bits of headroom because in two's complement the MSb serves as a sign bit). The headroom for a BFP vector is the *minimum* of headroom amongst each of its elements; in this case, 10 bits.

If we remove headroom from one mantissa of a BFP vector, all other mantissas must shift by the same number of bits, and the vector's exponent must be adjusted accordingly. A left-shift of one bit corresponds to reducing the exponent by 1, because a single bit left-shift corresponds to multiplication by 2.

In this case, if we remove 10 bits of headroom and subtract 10 from the exponent we get the following:

```
struct {
    int32_t mant[2];
    int32_t exp;
} vect = { { (1<<30), (0xFF >> 0) }, -10 };
```
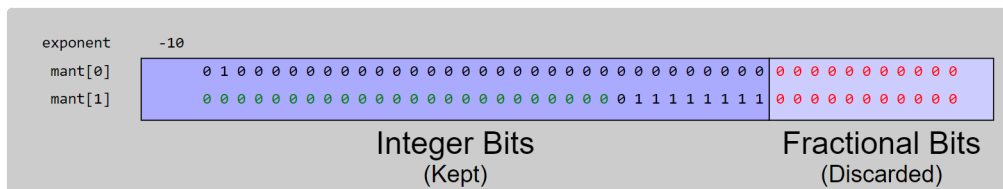

Now, no information is lost in either element. One of the main goals of BFP arithmetic is to keep the headroom in BFP vectors to the minimum necessary (equivalently, keeping the exponent as small as possible). That allows for maximum effective precision of the elements in the vector.

Note that the headroom of a vector also tells you something about the size of the largest magnitude mantissa in the vector. That information (in conjunction with exponents) can be used to determine the largest possible output of an operation without having to look at the mantissas.

For this reason, the BFP vectors in `lib_xcore_math` carry a field which tracks their current headroom. The functions in the BFP API use this property to make determinations about how best to preserve precision.

4 API Reference

4.1 XMath Types

Each of the main operand types used in this library has a short-hand which is used as a prefix in the naming of API operations. The following tables can be used for reference.

4.1.1 Common Vector Types

The following table indicates the types and abbreviations associated with various common vector types.

Table 4.1: Common Vector Types

Prefix	Object Type	Notes
`vect_s32`	`int32_t[]`	Raw vector of signed 32-bit integers.
`vect_s16`	`int16_t[]`	Raw vector of signed 16-bit integers.
`vect_s8`	`int8_t[]`	Raw vector of signed 8-bit integers.
`vect_complex_s32`	`complex_s32_t[]`	Raw vector of complex 32-bit integers.
`vect_complex_s16`	`(int16_t[], int16_t[])`	Complex 16-bit vectors are usually represented as a pair of 16-bit vectors. This is an optimization due to the word-alignment requirement when loading data into the VPU's vector registers.
`chunk_s32`	`int32_t[8]`	A 'chunk' is a fixed size vector corresponding to the size of the VPU vector registers.
`vect_qXX`	`int32_t[]`	When used in an API function name, the XX will be an actual number (e.g. `vect_q30_exp_small()`) indicating the fixed-point interpretation used by that function.
`vect_f32`	`float[]`	Raw vector of standard IEEE `float`
`vect_float_s32`	`float_s32_t[]`	Vector of non-standard 32-bit floating-point scalars.
`bfp_s32`	`bfp_s32_t`	Block floating-point vector contianing 32-bit mantissas.
`bfp_s16`	`bfp_s16_t`	Block floating-point vector contianing 16-bit mantissas.
`bfp_complex_s32`	`bfp_complex_s32_t`	Block floating-point vector contianing complex 32-bit mantissas.
`bfp_complex_s16`	`bfp_complex_s16_t`	Block floating-point vector contianing complex 16-bit mantissas.

4.1.2   Common Scalar Types

The following table indicates the types and abbreviations associated with various common scalar types.

Table 4.2: Common Scalar Types

Prefix	Object Type	Notes
s32	int32_t	32-bit signed integer. May be a simple integer, a fixed-point value or the mantissa of a floating-point value.
s16	int16_t	16-bit signed integer. May be a simple integer, a fixed-point value or the mantissa of a floating-point value.
s8	int8_t	8-bit signed integer. May be a simple integer, a fixed-point value or the mantissa of a floating-point value.
complex_s32	*complex_s32_t*	Signed complex integer with 32-bit real and 32-bit imaginary parts.
complex_s16	*complex_s16_t*	Signed complex integer with 16-bit real and 16-bit imaginary parts.
float_s64	*float_s64_t*	Non-standard floating-point scalar with exponent and 64-bit mantissa.
float_s32	*float_s32_t*	Non-standard floating-point scalar with exponent and 32-bit mantissa.
qXX	int32_t	32-bit fixed-point value with XX fractional bits (i.e. exponent of -XX).
f32	float	Standard IEEE 754 single-precision float.
f64	double	Standard IEEE 754 double-precision float.
float_complex_s64	*float_complex_s64_t*	Floating-point value with exponent and complex mantissa with 64-bit real and imaginary parts.
float_complex_s32	*float_complex_s32_t*	Floating-point value with exponent and complex mantissa with 32-bit real and imaginary parts.
float_complex_s16	*float_complex_s16_t*	Floating-point value with exponent and complex mantissa with 16-bit real and imaginary parts.
N/A	*exponent_t*	Represents an exponent p as in 2^p. Unless otherwise specified exponent are always assumed to have a base of 2.
N/A	*headroom_t*	The headroom of a scalar or vector. See *Headroom* for more information.
N/A	*right_shift_t*	Represents a rightward bit-shift of a certain number of bits. Care should be taken, as sometimes this is treated as unsigned.
N/A	*left_shift_t*	Represents a leftward bit-shift of a certain number of bits. Care should be taken, as sometimes this is treated as unsigned.

4.1.3 Block Floating-Point Types

group `type_bfp`

Enums

enum `bfp_flags_e`

(Opaque) Flags field for BFP vectors.

> **Warning:** Users should not manually modify fields of this type, as it is intended to be opaque.

Values:

enumerator `BFP_FLAG_DYNAMIC`

Indicates that BFP vector's mantissa buffer(s) were allocated dynamically

This flag lets the bfp_*_dealloc() functions know whether the mantissa vector must be free()ed.

struct `bfp_s32_t`

#include <types.h> A block floating-point vector of 32-bit elements.

Initialized with the `bfp_s32_init()` function.

The logical quantity represented by each element of this vector is: `data[i]*2^(exp)` where the multiplication and exponentiation are using real (non-modular) arithmetic.

The BFP API keeps the `hr` field up-to-date with the current headroom of `data[]` so as to minimize precision loss as elements become small.

Public Members

int32_t *`data`

Pointer to the underlying element buffer.

exponent_t `exp`

Exponent associated with the vector.

headroom_t `hr`

Current headroom in the `data[]`

unsigned `length`

Current size of `data[]`, expressed in elements

bfp_flags_e `flags`

BFP vector flags. Users should not normally modify these manually.

struct `bfp_s16_t`

#include <types.h> A block floating-point vector of 16-bit elements.

Initialized with the `bfp_s16_init()` function.

The logical quantity represented by each element of this vector is: `data[i] * 2^(exp)` where the multiplication and exponentiation are using real (non-modular) arithmetic.

The BFP API keeps the `hr` field up-to-date with the current headroom of `data[]` so as to minimize precision loss as elements become small. [*bfp_s16_t*]

Public Members

int16_t *__data__

> Pointer to the underlying element buffer.

exponent_t __exp__

> Exponent associated with the vector.

headroom_t __hr__

> Current headroom in the `data[]`

unsigned __length__

> Current size of `data[]`, expressed in elements

bfp_flags_e __flags__

> BFP vector flags. Users should not normally modify these manually.

struct __bfp_complex_s32_t__

> *#include <types.h>* [*bfp_s16_t*]

> A block floating-point vector of complex 32-bit elements.

> Initialized with the `bfp_complex_s32_init()` function.

> The logical quantity represented by each element of this vector is: `data[k].re * 2^(exp) + i * data[k].im * 2^(exp)` where the multiplication and exponentiation are using real (non-modular) arithmetic, and i is sqrt(-1)

> The BFP API keeps the `hr` field up-to-date with the current headroom of `data[]` so as to minimize precision loss as elements become small. [*bfp_complex_s32_t*]

Public Members

complex_s32_t *__data__

> Pointer to the underlying element buffer.

exponent_t __exp__

> Exponent associated with the vector.

headroom_t __hr__

> Current headroom in the `data[]`

unsigned **length**

> Current size of `data[]`, expressed in elements

bfp_flags_e **flags**

> BFP vector flags. Users should not normally modify these manually.

struct **bfp_complex_s16_t**

> *#include <types.h>* [*bfp_complex_s32_t*]
>
> A block floating-point vector of complex 16-bit elements.
>
> Initialized with the `bfp_complex_s16_init()` function.
>
> The logical quantity represented by each element of this vector is: `data[k].re * 2^(exp) + i * data[k].im * 2^(exp)` where the multiplication and exponentiation are using real (non-modular) arithmetic, and i is sqrt(-1)
>
> The BFP API keeps the `hr` field up-to-date with the current headroom of `data[]` so as to minimize precision loss as elements become small. [*bfp_complex_s16_t*]

Public Members

int16_t ***real**

> Pointer to the underlying element buffer.

int16_t ***imag**

> Pointer to the underlying element buffer.

exponent_t **exp**

> Exponent associated with the vector.

headroom_t **hr**

> Current headroom in the `data[]`

unsigned **length**

> Current size of `data[]`, expressed in elements

bfp_flags_e **flags**

> BFP vector flags. Users should not normally modify these manually.

4.1.4 Scalar Types (Integer)

group **type_scalar_int**

Typedefs

typedef int **exponent_t**

    An exponent.

    Many places in this API make use of integers representing the exponent associated with some floating-point value or block floating-point vector.

    For a floating-point value $x \cdot 2^p$, p is the exponent, and may usually be positive or negative.

typedef unsigned **headroom_t**

    Headroom of some integer or integer array.

    Represents the headroom of a signed or unsigned integer, complex integer or channel pair, or the headroom of the mantissa array of a block floating-point vector.

typedef int **right_shift_t**

    A rightwards arithmetic bit-shift.

    Represents a right bit-shift to be applied to an integer. May be signed or unsigned, depending on context. If signed, negative values represent leftward bit-shifts.

    See also:

    left_shift_t

typedef int **left_shift_t**

    A leftwards arithmetic bit-shift.

    Represents a left bit-shift to be applied to an integer. May be signed or unsigned, depending on context. If signed, negative values represent rightward bit-shifts.

    See also:

    right_shift_t

struct **complex_s64_t**

    #include <types.h> A complex number with a 64-bit real part and 64-bit imaginary part.

Public Members

int64_t **re**

    Real Part.

int64_t **im**

    Imaginary Part.

struct **complex_s32_t**

    #include <types.h> A complex number with a 32-bit real part and 32-bit imaginary part.

Public Members

int32_t **re**

    Real Part.

int32_t **im**

    Imaginary Part.

struct **complex_s16_t**

    #include <types.h> A complex number with a 16-bit real part and 16-bit imaginary part.

Public Members

int16_t **re**

    Real Part.

int16_t **im**

    Imaginary Part.

4.1.5   Scalar Types (Floating-Point)

group **type_scalar_float**

struct **float_s32_t**

    #include <types.h> A floating-point scalar with a 32-bit mantissa.

    Represents a (non-standard) floating-point value given by $M \cdot 2^x$, where M is the 32-bit mantissa `mant`, and x is the exponent `exp`.

    To convert a *float_s32_t* to a standard IEEE754 single-precision floating-point value (which may result in a loss of precision):

```
float to_ieee_float(float_s32_t x) {
    return ldexpf(x.mant, x.exp);
}
```

Public Members

int32_t **mant**

    32-bit mantissa

exponent_t **exp**

    exponent

struct **float_s64_t**

    #include <types.h> A floating-point scalar with a 64-bit mantissa.

Represents a (non-standard) floating-point value given by $M \cdot 2^x$, where M is the 64-bit mantissa `mant`, and x is the exponent `exp`.

To convert a *float_s64_t* to a standard IEEE754 double-precision floating-point value (which may result in a loss of precision):

```
double to_ieee_float(float_s64_t x) {
    return ldexp(x.mant, x.exp);
}
```

Public Members

int64_t **mant**
> 64-bit mantissa

exponent_t **exp**
> exponent

struct **float_complex_s16_t**
> *#include <types.h>* A complex floating-point scalar with a complex 16-bit mantissa.
>
> Represents a (non-standard) complex floating-point value given by $A + j \cdot B \cdot 2^x$, where A is `mant.re`, the 16-bit real part of the mantissa, B is `mant.im`, the 16-bit imaginary part of the mantissa, and x is the exponent `exp`.

Public Members

complex_s16_t **mant**
> complex 16-bit mantissa

exponent_t **exp**
> exponent

struct **float_complex_s32_t**
> *#include <types.h>* A complex floating-point scalar with a complex 32-bit mantissa.
>
> Represents a (non-standard) complex floating-point value given by $A + j \cdot B \cdot 2^x$, where A is `mant.re`, the 32-bit real part of the mantissa, B is `mant.im`, the 32-bit imaginary part of the mantissa, and x is the exponent `exp`.

Public Members

complex_s32_t **mant**
> complex 32-bit mantissa

exponent_t **exp**
> exponent

struct **float_complex_s64_t**

#include <types.h> A complex floating-point scalar with a complex 64-bit mantissa.

Represents a (non-standard) complex floating-point value given by $A + j \cdot B \cdot 2^x$, where A is `mant.re`, the 64-bit real part of the mantissa, B is `mant.im`, the 64-bit imaginary part of the mantissa, and x is the exponent `exp`.

Public Members

complex_s64_t **mant**
> complex 64-bit mantissa

exponent_t **exp**
> exponent

struct **complex_float_t**

#include <types.h> [*bfp_complex_s16_t*]

A complex number with a single-precision floating-point real part and a single-precision floating-point imaginary part.

Public Members

float **re**
> Real Part.

float **im**
> Imaginary Part.

struct **complex_double_t**

#include <types.h> A complex number with a double-precision floating-point real part and a double-precision floating-point imaginary part.

Public Members

double **re**
> Real Part.

double **im**
> Imaginary Part.

4.1.6   Scalar Types (Fixed-Point)

group `type_scalar_fixed`

Typedefs

typedef int32_t **q1_31**

> Q1.31 (Signed) Fixed-point value.
>
> Represents a signed, 32-bit, real, fixed-point value with 31 fractional bits (i.e. an implicit exponent of -31).
>
> Capable of representing values in the range $[-1.0, 1.0)$

typedef int32_t **q2_30**

> Q2.30 (Signed) Fixed-point value.
>
> Represents a signed, 32-bit, real, fixed-point value with 30 fractional bits (i.e. an implicit exponent of -30).
>
> Capable of representing values in the range $[-2.0, 2.0)$

typedef int32_t **q4_28**

> Q4.28 (Signed) Fixed-point value.
>
> Represents a signed, 32-bit, real, fixed-point value with 28 fractional bits (i.e. an implicit exponent of -28).
>
> Capable of representing values in the range $[-8.0, 8.0)$

typedef int32_t **q8_24**

> Q8.24 (Signed) Fixed-point value.
>
> Represents a signed, 32-bit, real, fixed-point value with 24 fractional bits (i.e. an implicit exponent of -24).
>
> Capable of representing values in the range $[-128.0, 128.0)$

typedef uint32_t **uq0_32**

> UQ0.32 (Unsigned) Fixed-point value.
>
> Represents an unsigned, 32-bit, real, fixed-point value with 32 fractional bits (i.e. an implicit exponent of -32).
>
> Capable of representing values in the range $[0, 1.0)$

typedef uint32_t **uq1_31**

> UQ1.31 (Unsigned) Fixed-point value.
>
> Represents an unsigned, 32-bit, real, fixed-point value with 31 fractional bits (i.e. an implicit exponent of -31).
>
> Capable of representing values in the range $[0, 2.0)$

typedef uint32_t **uq2_30**

> UQ2.30 (Unsigned) Fixed-point value.

Represents an unsigned, 32-bit, real, fixed-point value with 30 fractional bits (i.e. an implicit exponent of -30).

Capable of representing values in the range $[0, 4.0)$

typedef uint32_t **uq4_28**

UQ4.28 (Unsigned) Fixed-point value.

Represents an unsigned, 32-bit, real, fixed-point value with 28 fractional bits (i.e. an implicit exponent of -28).

Capable of representing values in the range $[0, 16.0)$

typedef uint32_t **uq8_24**

UQ8.24 (Unsigned) Fixed-point value.

Represents an unsigned, 32-bit, real, fixed-point value with 24 fractional bits (i.e. an implicit exponent of -24).

Capable of representing values in the range $[0, 256.0)$

typedef *q1_31* **sbrad_t**

Specialized angular unit used by this library.

'sbrad' is a kind of modified binary radian (hence 'brad') which takes into account the symmetries of $sin(\theta)$.

Use *radians_to_sbrads()* to convert from radians to `sbrad_t`.

typedef *q8_24* **radian_q24_t**

Angle measurement in radians using a Q8.24 representation.

4.1.7 Misc Types

group **type_misc**

struct **split_acc_s32_t**

#include <types.h> Holds a set of sixteen 32-bit accumulators in the XS3 VPU's internal format.

The XS3 VPU stores 32-bit accumulators with the most significant 16-bits stored in one 256-bit vector register (called vD), and the least significant 16-bit stored in another 256-bit register (called vR). This struct reflects that internal format, and is occasionally used to store intermediate results.

Note: `vR` is unsigned. This reflects the fact that a signed 16-bit integer `0xSTUVWXYZ` is always exactly `0x0000WXYZ` larger than `0xSTUV0000`. To combine the upper and lower 16-bits of an accumulator, use `(((int32_t)vD[k]) << 16) + vR[k]`.

---

int16_t **vD**[16]

    Most significant 16 bits of accumulators.

uint16_t **vR**[16]

    Least significant 16 bits of accumulators.

4.2   Block Floating-Point API

4.2.1   BFP API Quick Reference

The tables below list the functions of the block floating-point API. The "EW" column indicates whether the operation acts element-wise.

The "Signature" column is intended as a hint which quickly conveys the kind of the conceptual inputs to and outputs from the operation. The signatures are only intended to convey how many (conceptual) inputs and outputs there are, and their dimensionality.

The functions themselves will typically take more arguments than these signatures indicate. Check the function's full documentation to get more detailed information.

The following symbols are used in the signatures:

Symbol	Description
$\mathbb{S}$	A scalar input or output value.
$\mathbb{V}$	A vector-valued input or output.
$\mathbb{M}$	A matrix-valued input or output.
$\varnothing$	Placeholder indicating no input or output.

For example, the operation signature $(\mathbb{V} \times \mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$ indicates the operation takes two vector inputs and a scalar input, and the output is a vector.

- *32-Bit BFP Ops*

- *16-Bit BFP Ops*

- *Complex 32-Bit BFP Ops*

- *Complex 16-Bit BFP Ops*

4.2.1.1   32-Bit BFP API Quick Reference

<p align="center">Table 4.4: 32-Bit BFP API - Quick Reference</p>

Function	EW	Signature	Brief
`bfp_s32_init()`		$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$	Initialize (static)
`bfp_s32_alloc()`		$\varnothing \to \mathbb{V}$	Initialize (dynamic)
`bfp_s32_dealloc()`		$\mathbb{V} \to \varnothing$	Deinitialize
`bfp_s32_set()`	x	$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$	Set All Elements
`bfp_s32_use_exponent()`		$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$	Force Exponent
`bfp_s32_headroom()`		$\mathbb{V} \to \mathbb{S}$	Get Headroom
`bfp_s32_shl()`	x	$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$	Shift Mantissas
`bfp_s32_add()`	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Add Vector
`bfp_s32_add_scalar()`		$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$	Add Scalar
`bfp_s32_sub()`	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Subtract Vector
`bfp_s32_mul()`	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Multiply Vector
`bfp_s32_macc()`	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Multiply-Accumulate
`bfp_s32_nmacc()`	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Negated Multiply-Accumulate
`bfp_s32_scale()`		$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$	Multiply Scalar
`bfp_s32_abs()`	x	$\mathbb{V} \to \mathbb{V}$	Absolute Values
`bfp_s32_sum()`		$\mathbb{V} \to \mathbb{S}$	Sum Elements
`bfp_s32_dot()`		$(\mathbb{V} \times \mathbb{V}) \to \mathbb{S}$	Inner Product
`bfp_s32_clip()`	x	$(\mathbb{V} \times \mathbb{S} \times \mathbb{S}) \to \mathbb{V}$	Clip Bounds
`bfp_s32_rect()`	x	$\mathbb{V} \to \mathbb{V}$	Rectify Elements
`bfp_s32_to_bfp_s16()`		$\mathbb{V} \to \mathbb{V}$	Convert to 16-bit
`bfp_s32_sqrt()`	x	$\mathbb{V} \to \mathbb{V}$	Square Root
`bfp_s32_inverse()`	x	$\mathbb{V} \to \mathbb{V}$	Multiplicative Inverse
`bfp_s32_abs_sum()`		$\mathbb{V} \to \mathbb{S}$	Absolute Sum Elements
`bfp_s32_mean()`		$\mathbb{V} \to \mathbb{S}$	Vector Mean Value
`bfp_s32_energy()`		$\mathbb{V} \to \mathbb{S}$	Vector Energy
`bfp_s32_rms()`		$\mathbb{V} \to \mathbb{S}$	Vector RMS Value
`bfp_s32_max()`		$\mathbb{V} \to \mathbb{S}$	Vector Max Element
`bfp_s32_min()`		$\mathbb{V} \to \mathbb{S}$	Vector Min Element
`bfp_s32_max_elementwise()`	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Elementwise Max
`bfp_s32_min_elementwise()`	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Elementwise Min
`bfp_s32_argmax()`		$\mathbb{V} \to \mathbb{S}$	Max Element Index
`bfp_s32_argmin()`		$\mathbb{V} \to \mathbb{S}$	Min Element Index
`bfp_s32_convolve_valid()`		$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Convolve With Kernel (Valid mode)
`bfp_s32_convolve_same()`		$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Convolve With Kernel (Same mode)

4.2.1.2   16-Bit BFP API Quick Reference

Table 4.5: 16-Bit BFP API - Quick Reference

Function	EW	Signature	Brief
`bfp_s16_init()`		$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$	Initialize (static)
`bfp_s16_alloc()`		$\varnothing \rightarrow \mathbb{V}$	Initialize (dynamic)
`bfp_s16_dealloc()`		$\mathbb{V} \rightarrow \varnothing$	Deinitialize
`bfp_s16_set()`	x	$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$	Set All Elements
`bfp_s16_use_exponent()`		$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$	Force Exponent
`bfp_s16_headroom()`		$\mathbb{V} \rightarrow \mathbb{S}$	Get Headroom
`bfp_s16_shl()`	x	$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$	Shift Mantissas
`bfp_s16_add()`	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Add Vector
`bfp_s16_add_scalar()`		$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$	Add Scalar
`bfp_s16_sub()`	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Subtract Vector
`bfp_s16_mul()`	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Multiply Vector
`bfp_s16_macc()`	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Multiply-Accumulate
`bfp_s16_nmacc()`	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Negated Multiply-Accumulate
`bfp_s16_scale()`		$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$	Multiply Scalar
`bfp_s16_abs()`	x	$\mathbb{V} \rightarrow \mathbb{V}$	Absolute Values
`bfp_s16_sum()`		$\mathbb{V} \rightarrow \mathbb{S}$	Sum Elements
`bfp_s16_dot()`		$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{S}$	Inner Product
`bfp_s16_clip()`	x	$(\mathbb{V} \times \mathbb{S} \times \mathbb{S}) \rightarrow \mathbb{V}$	Clip Bounds
`bfp_s16_rect()`	x	$\mathbb{V} \rightarrow \mathbb{V}$	Rectify Elements
`bfp_s16_to_bfp_s32()`	x	$\mathbb{V} \rightarrow \mathbb{V}$	Convert to 32-bit
`bfp_s16_sqrt()`	x	$\mathbb{V} \rightarrow \mathbb{V}$	Square Root
`bfp_s16_inverse()`	x	$\mathbb{V} \rightarrow \mathbb{V}$	Multiplicative Inverse
`bfp_s16_abs_sum()`		$\mathbb{V} \rightarrow \mathbb{V}$	Absolute Sum Elements
`bfp_s16_mean()`		$\mathbb{V} \rightarrow \mathbb{V}$	Vector Mean Value
`bfp_s16_energy()`		$\mathbb{V} \rightarrow \mathbb{S}$	Vector Energy
`bfp_s16_rms()`		$\mathbb{V} \rightarrow \mathbb{S}$	Vector RMS Value
`bfp_s16_max()`		$\mathbb{V} \rightarrow \mathbb{S}$	Vector Max Element
`bfp_s16_min()`		$\mathbb{V} \rightarrow \mathbb{S}$	Vector Min Element
`bfp_s16_max_elementwise()`	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Elementwise Max
`bfp_s16_min_elementwise()`	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Elementwise Min
`bfp_s16_argmax()`		$\mathbb{V} \rightarrow \mathbb{S}$	Max Element Index
`bfp_s16_argmin()`		$\mathbb{V} \rightarrow \mathbb{S}$	Min Element Index
`bfp_s16_accumulate()`	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Elementwise Accumulate

4.2.1.3 Complex 32-bit BFP API Quick Reference

Table 4.6: Complex 32-Bit BFP API - Quick Reference

Function	EW	Signature	Brief
`bfp_complex_s32_init()`		$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$	Initialize (static)
`bfp_complex_s32_alloc()`		$\varnothing \rightarrow \mathbb{V}$	Initialize (dynamic)
`bfp_complex_s32_dealloc()`		$\mathbb{V} \rightarrow \varnothing$	Deinitialize
`bfp_complex_s32_set()`	x	$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$	Set All Elements
`bfp_complex_s32_use_exponent()`		$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$	Force Exponent
`bfp_complex_s32_headroom()`		$\mathbb{V} \rightarrow \mathbb{S}$	Get Headroom
`bfp_complex_s32_shl()`	x	$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$	Shift Mantissas
`bfp_complex_s32_real_mul()`	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Real Vector Multiply
`bfp_complex_s32_mul()`	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Complex Vector Multiply
`bfp_complex_s32_conj_mul()`	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Complex Vector Conjugate Multiply
`bfp_complex_s32_macc()`	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Complex Vector Multiply-Accumulate
`bfp_complex_s32_nmacc()`	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Complex Vector Negated Multiply-Accumulate
`bfp_complex_s32_conj_macc()`	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Complex Vector Conjugate Multiply-Accumulate
`bfp_complex_s32_conj_nmacc()`	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Complex Vector Negated Conjugate Multiply-Accumulate
`bfp_complex_s32_real_scale()`		$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$	Real Scalar Multiply
`bfp_complex_s32_scale()`		$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$	Complex Scalar Multiply
`bfp_complex_s32_add()`	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Complex Vector Add
`bfp_complex_s32_add_scalar()`		$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$	Complex Scalar Add
`bfp_complex_s32_sub()`		$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Complex Vector Subtract
`bfp_complex_s32_to_bfp_complex_s16()`	x	$\mathbb{V} \rightarrow \mathbb{V}$	Convert to 16-bit
`bfp_complex_s32_squared_mag()`	x	$\mathbb{V} \rightarrow \mathbb{V}$	Squared Magnitude
`bfp_complex_s32_mag()`	x	$\mathbb{V} \rightarrow \mathbb{V}$	Magnitude
`bfp_complex_s32_sum()`		$\mathbb{V} \rightarrow \mathbb{S}$	Vector Sum
`bfp_complex_s32_conjugate()`	x	$\mathbb{V} \rightarrow \mathbb{V}$	Complex Conjugate
`bfp_complex_s32_energy()`		$\mathbb{V} \rightarrow \mathbb{S}$	Vector Energy
`bfp_complex_s32_make()`	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$	Construct Complex From Real and Imaginary
`bfp_complex_s32_real_part()`	x	$\mathbb{V} \rightarrow \mathbb{V}$	Real Part
`bfp_complex_s32_imag_part()`	x	$\mathbb{V} \rightarrow \mathbb{V}$	Imaginary Part

4.2.1.4 Complex 16-bit BFP API Quick Reference

Table 4.7: Complex 16-Bit BFP API - Quick Reference

Function	EW	Signature	Brief
`bfp_complex_s16_init()`		$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$	Initialize (static)
`bfp_complex_s16_alloc()`		$\varnothing \to \mathbb{V}$	Initialize (dynamic)
`bfp_complex_s16_dealloc()`		$\mathbb{V} \to \varnothing$	Deinitialize
`bfp_complex_s16_set()`	x	$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$	Set All Elements
`bfp_complex_s16_use_exponent()`		$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$	Force Exponent
`bfp_complex_s16_headroom()`		$\mathbb{V} \to \mathbb{S}$	Get Headroom
`bfp_complex_s16_shl()`	x	$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$	Shift Mantissas
`bfp_complex_s16_real_mul()`	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Real Vector Multiply
`bfp_complex_s16_mul()`	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Complex Vector Multiply
`bfp_complex_s16_conj_mul()`	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Complex Vector Conjugate Multiply
`bfp_complex_s16_macc()`	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Complex Vector Multiply-Accumulate
`bfp_complex_s16_nmacc()`	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Complex Vector Negated Multiply-Accumulate
`bfp_complex_s16_conj_macc()`	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Complex Vector Conjugate Multiply-Accumulate
`bfp_complex_s16_conj_nmacc()`	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Complex Vector Negated Conjugate Multiply-Accumulate
`bfp_complex_s16_real_scale()`		$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$	Real Scalar Multiply
`bfp_complex_s16_scale()`		$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$	Complex Scalar Multiply
`bfp_complex_s16_add()`	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Complex Vector Add
`bfp_complex_s16_add_scalar()`		$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$	Complex Scalar Add
`bfp_complex_s16_sub()`		$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$	Complex Vector Subtract
`bfp_complex_s16_to_bfp_complex_s32()`	x	$\mathbb{V} \to \mathbb{V}$	Convert to 32-bit
`bfp_complex_s16_squared_mag()`	x	$\mathbb{V} \to \mathbb{V}$	Squared Magnitude
`bfp_complex_s16_sum()`		$\mathbb{V} \to \mathbb{S}$	Vector Sum
`bfp_complex_s16_mag()`	x	$\mathbb{V} \to \mathbb{V}$	Magnitude
`bfp_complex_s16_conjugate()`	x	$\mathbb{V} \to \mathbb{V}$	Complex Conjugate
`bfp_complex_s16_energy()`		$\mathbb{V} \to \mathbb{S}$	Vector Energy

4.2.2   16-Bit Block Floating-Point API

group `bfp_s16_api`

Functions

void **bfp_s16_init**(*bfp_s16_t* *a, int16_t *data, const *exponent_t* exp, const unsigned length, const unsigned calc_hr)

Initialize a 16-bit BFP vector.

This function initializes each of the fields of BFP vector `a`.

`data` points to the memory buffer used to store elements of the vector, so it must be at least `length * 2` bytes long, and must begin at a word-aligned address.

`exp` is the exponent assigned to the BFP vector. The logical value associated with the `kth` element of the vector after initialization is $data_k \cdot 2^{exp}$.

If `calc_hr` is false, `a->hr` is initialized to 0. Otherwise, the headroom of the the BFP vector is calculated and used to initialize `a->hr`.

> **Parameters**
> - `a` – **[out]** BFP vector to initialize
> - `data` – **[in]** `int16_t` buffer used to back `a`
> - `exp` – **[in]** Exponent of BFP vector
> - `length` – **[in]** Number of elements in the BFP vector
> - `calc_hr` – **[in]** Boolean indicating whether the HR of the BFP vector should be calculated

bfp_s16_t **bfp_s16_alloc**(const unsigned length)

Dynamically allocate a 16-bit BFP vector from the heap.

If allocation was unsuccessful, the `data` field of the returned vector will be NULL, and the `length` field will be zero. Otherwise, `data` will point to the allocated memory and the `length` field will be the user-specified length. The `length` argument must not be zero.

Neither the BFP exponent, headroom, nor the elements of the allocated mantissa vector are set by this function. To set the BFP vector elements to a known value, use *bfp_s16_set()* on the retuned BFP vector.

BFP vectors allocated using this function must be deallocated using *bfp_s16_dealloc()* to avoid a memory leak.

To initialize a BFP vector using static memory allocation, use *bfp_s16_init()* instead.

> **See also:**
>
> *bfp_s16_dealloc*, *bfp_s16_init*

> **Note:**   Dynamic allocation of BFP vectors relies on allocation from the heap, and offers no guarantees about the execution time. Use of this function in any time-critical section of code is highly discouraged.

> **Parameters**
> - `length` – **[in]** The length of the BFP vector to be allocated (in elements)

Returns
> 16-bit BFP vector

void **bfp_s16_dealloc**(*bfp_s16_t* *vector)
> Deallocate a 16-bit BFP vector allocated by *bfp_s16_alloc()*.
>
> Use this function to free the heap memory allocated by *bfp_s16_alloc()*.
>
> BFP vectors whose mantissa buffer was (successfully) dynamically allocated have a flag set which indicates as much. This function can safely be called on any *bfp_s16_t* which has not had its `flags` or `data` manually manipulated, including:
>
> - *bfp_s16_t* resulting from a successful call to *bfp_s16_alloc()*
> - *bfp_s16_t* resulting from an unsuccessful call to *bfp_s16_alloc()*
> - *bfp_s16_t* initialized with a call to *bfp_s16_init()*
>
> In the latter two cases, this function does nothing. In the former, the `data`, `length` and `flags` fields of `vector` are cleared to zero.

> **See also:**
>
> *bfp_s16_alloc*
>
> > **Parameters**
> >
> > - `vector` – **[in]** BFP vector to be deallocated.

void **bfp_s16_set**(*bfp_s16_t* *a, const int16_t b, const *exponent_t* exp)
> Set all elements of a 16-bit BFP vector to a specified value.
>
> The exponent of `a` is set to `exp`, and each element's mantissa is set to `b`.
>
> After performing this operation, all elements will represent the same value $b \cdot 2^{exp}$.
>
> `a` must have been initialized (see *bfp_s16_init()*).
>
> > **Parameters**
> >
> > - `a` – **[out]** BFP vector to update
> > - `b` – **[in]** New value each mantissa is set to
> > - `exp` – **[in]** New exponent for the BFP vector

headroom_t **bfp_s16_headroom**(*bfp_s16_t* *b)
> Get the headroom of a 16-bit BFP vector.
>
> The headroom of a vector is the number of bits its elements can be left-shifted without losing any information. It conveys information about the range of values that vector may contain, which is useful for determining how best to preserve precision in potentially lossy block floating-point operations.
>
> In a BFP context, headroom applies to mantissas only, not exponents.
>
> In particular, if the 16-bit mantissa vector $\bar{x}$ has N bits of headroom, then for any element x_k of $\bar{x}$
>
> $$-2^{15-N} \le x_k < 2^{15-N}$$
>
> And for any element $X_k = x_k \cdot 2^{x\_exp}$ of a complex BFP vector $\bar{X}$
>
> $$-2^{15+x\_exp-N} \le X_k < 2^{15+x\_exp-N}$$
>
> This function determines the headroom of `b`, updates `b->hr` with that value, and then returns `b->hr`.
>
> > **Parameters**
> >
> > - `b` – BFP vector to get the headroom of

void **bfp_s16_use_exponent**(*bfp_s16_t* *a, const *exponent_t* exp)

Modify a 16-bit BFP vector to use a specified exponent.

This function forces BFP vector $\bar{A}$ to use a specified exponent. The mantissa vector $\bar{a}$ will be bit-shifted left or right to compensate for the changed exponent.

This function can be used, for example, before calling a fixed-point arithmetic function to ensure the underlying mantissa vector has the needed Q-format. As another example, this may be useful when communicating with peripheral devices (e.g. via I2S) that require sample data to be in a specified format.

Note that this sets the *current* encoding, and does not *fix* the exponent permanently (i.e. subsequent operations may change the exponent as usual).

If the required fixed-point Q-format is QX.Y, where Y is the number of fractional bits in the resulting mantissas, then the associated exponent (and value for parameter exp) is -Y.

a points to input BFP vector $\bar{A}$, with mantissa vector $\bar{a}$ and exponent a_exp. a is updated in place to produce resulting BFP vector $\bar{\bar{A}}$ with mantissa vector $\bar{\bar{a}}$ and exponent $\tilde{a}\_exp$.

exp is $\tilde{a}\_exp$, the required exponent. $\Delta p = \tilde{a}\_exp - a\_exp$ is the required change in exponent.

If $\Delta p = 0$, the BFP vector is left unmodified.

If $\Delta p > 0$, the required exponent is larger than the current exponent and an arithmetic right-shift of Δp bits is applied to the mantissas $\bar{a}$. When applying a right-shift, precision may be lost by discarding the Δp least significant bits.

If $\Delta p < 0$, the required exponent is smaller than the current exponent and a left-shift of Δp bits is applied to the mantissas $\bar{a}$. When left-shifting, saturation logic will be applied such that any element that can't be represented exactly with the new exponent will saturate to the 16-bit saturation bounds.

The exponent and headroom of a are updated by this function.

Operation Performed

$$\Delta p = \tilde{a}\_exp - a\_exp$$
$$\tilde{a_k} \leftarrow sat_{16}(a_k \cdot 2^{-\Delta p})$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{A} \text{ (in elements)}$$

Parameters

- **a** − **[inout]** Input BFP vector $\bar{A}$ / Output BFP vector $\bar{\bar{A}}$
- **exp** − **[in]** The required exponent, $\tilde{a}\_exp$

void **bfp_s16_shl**(*bfp_s16_t* *a, const *bfp_s16_t* *b, const *left_shift_t* b_shl)

Apply a left-shift to the mantissas of a 16-bit BFP vector.

Each mantissa of input BFP vector $\bar{B}$ is left-shifted b_shl bits and stored in the corresponding element of output BFP vector $\bar{A}$.

This operation can be used to add or remove headroom from a BFP vector.

b_shl is the number of bits that each mantissa will be left-shifted. This shift is signed and arithmetic, so negative values for b_shl will right-shift the mantissas.

a and b must have been initialized (see *bfp_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Note that this operation bypasses the logic protecting the caller from saturation or underflows. Output values saturate to the symmetric 16-bit range (the open interval $(-2^{15}, 2^{15})$). To avoid saturation, b_shl should be no greater than the headroom of b (b->hr).

Operation Performed

$$a_k \leftarrow sat_{16}(\lfloor b_k \cdot 2^{b\_shl} \rfloor)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$
$$\text{and } b_k \text{ and } a_k \text{ are the } k\text{th mantissas from } \bar{B} \text{ and } \bar{A} \text{ respectively}$$

Parameters

- **a** – **[out]** Output BFP vector $\bar{A}$
- **b** – **[in]** Input BFP vector $\bar{B}$
- **b_shl** – **[in]** Signed arithmetic left-shift to be applied to mantissas of $\bar{B}$.

void **bfp_s16_add**(*bfp_s16_t* *a, const *bfp_s16_t* *b, const *bfp_s16_t* *c)

Add two 16-bit BFP vectors together.

Add together two input BFP vectors $\bar{B}$ and $\bar{C}$ and store the result in BFP vector $\bar{A}$.

a, b and c must have been initialized (see *bfp_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b or c.

Operation Performed

$$\bar{A} \leftarrow \bar{B} + \bar{C}$$

Parameters

- **a** – **[out]** Output BFP vector $\bar{A}$
- **b** – **[in]** Input BFP vector $\bar{B}$
- **c** – **[in]** Input BFP vector $\bar{C}$

void **bfp_s16_add_scalar**(*bfp_s16_t* *a, const *bfp_s16_t* *b, const float c)

Add a scalar to a 16-bit BFP vector.

Add a real scalar c to input BFP vector $\bar{B}$ and store the result in BFP vector $\bar{A}$.

a, and b must have been initialized (see *bfp_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$\bar{A} \leftarrow \bar{B} + c$$

Parameters

- **a** – **[out]** Output BFP vector $\bar{A}$
- **b** – **[in]** Input BFP vector $\bar{B}$
- **c** – **[in]** Input scalar c

void **bfp_s16_sub**(*bfp_s16_t* *a, const *bfp_s16_t* *b, const *bfp_s16_t* *c)

Subtract one 16-bit BFP vector from another.

Subtract input BFP vector $\bar{C}$ from input BFP vector $\bar{C}$ and store the result in BFP vector $\bar{A}$.

a, b and c must have been initialized (see *bfp_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b or c.

Operation Performed

$$\bar{A} \leftarrow \bar{B} - \bar{C}$$

Parameters

- **a** − **[out]** Output BFP vector $\bar{A}$
- **b** − **[in]** Input BFP vector $\bar{B}$
- **c** − **[in]** Input BFP vector $\bar{C}$

void **bfp_s16_mul**(*bfp_s16_t* *a, const *bfp_s16_t* *b, const *bfp_s16_t* *c)

Multiply one 16-bit BFP vector by another element-wise.

Multiply each element of input BFP vector $\bar{B}$ by the corresponding element of input BFP vector $\bar{C}$ and store the results in output BFP vector $\bar{A}$.

a, b and c must have been initialized (see *bfp_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b or c.

Operation Performed

$$A_k \leftarrow B_k \cdot C_k$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **a** − Output BFP vector $\bar{A}$
- **b** − Input BFP vector $\bar{B}$
- **c** − Input BFP vector $\bar{C}$

void **bfp_s16_macc**(*bfp_s16_t* *acc, const *bfp_s16_t* *b, const *bfp_s16_t* *c)

Multiply one 16-bit BFP vector by another element-wise and add the result to a third vector.

Operation Performed

$$A_k \leftarrow A_k + B_k \cdot C_k$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **acc** − **[inout]** Input/Output accumulator BFP vector $\bar{A}$
- **b** − **[in]** Input BFP vector $\bar{B}$
- **c** − **[in]** Input BFP vector $\bar{C}$

void **bfp_s16_nmacc**(*bfp_s16_t* *acc, const *bfp_s16_t* *b, const *bfp_s16_t* *c)

Multiply one 16-bit BFP vector by another element-wise and subtract the result from a third vector.

Operation Performed

$$A_k \leftarrow A_k - B_k \cdot C_k$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **acc** − **[inout]** Input/Output accumulator BFP vector $\bar{A}$
- **b** − **[in]** Input BFP vector $\bar{B}$
- **c** − **[in]** Input BFP vector $\bar{C}$

void **bfp_s16_scale**(*bfp_s16_t* *a, const *bfp_s16_t* *b, const float alpha)

Multiply a 16-bit BFP vector by a scalar.

Multiply input BFP vector $\bar{B}$ by scalar $\alpha \cdot 2^{\alpha\text{-}exp}$ and store the result in output BFP vector $\bar{A}$.

a and b must have been initialized (see *bfp_s16_init()*), and must be the same length.

alpha represents the scalar $\alpha \cdot 2^{\alpha\text{-}exp}$, where α is alpha.mant and α_exp is alpha.exp.

This operation can be performed safely in-place on b.

Operation Performed

$$\bar{A} \leftarrow \bar{B} \cdot (\alpha \cdot 2^{\alpha\_exp})$$

Parameters

- **a** − **[out]** Output BFP vector $\bar{A}$
- **b** − **[in]** Input BFP vector $\bar{B}$
- **alpha** − **[in]** Scalar by which $\bar{B}$ is multiplied

void **bfp_s16_abs**(*bfp_s16_t* *a, const *bfp_s16_t* *b)

Get the absolute values of elements of a 16-bit BFP vector.

Compute the absolute value of each element B_k of input BFP vector $\bar{B}$ and store the results in output BFP vector $\bar{A}$.

a and b must have been initialized (see *bfp_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$A_k \leftarrow |B_k|$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **a** − **[out]** Output BFP vector $\bar{A}$
- **b** − **[in]** Input BFP vector $\bar{B}$

float_s32_t **bfp_s16_sum**(const *bfp_s16_t* *b)

Sum the elements of a 16-bit BFP vector.

Sum the elements of input BFP vector $\bar{B}$ to get a result $A = a \cdot 2^{a\_exp}$, which is returned. The returned value has a 32-bit mantissa.

b must have been initialized (see *bfp_s16_init()*).

Operation Performed

$$A \leftarrow \sum_{k=0}^{N-1} (B_k)$$

where N is the length of $\bar{B}$

Parameters

- **b** − **[in]** Input BFP vector $\bar{B}$

Returns

A, the sum of elements of $\bar{B}$

float_s64_t **bfp_s16_dot**(const *bfp_s16_t* *b, const *bfp_s16_t* *c)

Compute the inner product of two 16-bit BFP vectors.

Adds together the element-wise products of input BFP vectors $\bar{B}$ and $\bar{C}$ for a result $A = a \cdot 2^{a\_exp}$, where a is the 64-bit mantissa of the result and a_exp is its associated exponent. A is returned.

b and c must have been initialized (see *bfp_s16_init()*), and must be the same length.

Operation Performed

$$a \cdot 2^{a\_exp} \leftarrow \sum_{k=0}^{N-1} (B_k \cdot C_k)$$

where N is the length of $\bar{B}$ and $\bar{C}$

Parameters

- **b** − **[in]** Input BFP vector $\bar{B}$
- **c** − **[in]** Input BFP vector $\bar{C}$

Returns

A, the inner product of vectors $\bar{B}$ and $\bar{C}$

void **bfp_s16_clip**(*bfp_s16_t* *a, const *bfp_s16_t* *b, const int16_t lower_bound, const int16_t upper_bound, const int bound_exp)

Clamp the elements of a 16-bit BFP vector to a specified range.

Each element A_k of output BFP vector $\bar{A}$ is set to the corresponding element B_k of input BFP vector $\bar{B}$ if it is in the range $[L \cdot 2^{bound\_exp}, U \cdot 2^{bound\_exp}]$, otherwise it is set to the nearest value inside that range.

a and b must have been initialized (see *bfp_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$A_k \leftarrow \begin{cases} L \cdot 2^{bound\_exp} & B_k < L \cdot 2^{bound\_exp} \\ U \cdot 2^{bound\_exp} & B_k > U \cdot 2^{bound\_exp} \\ B_k & otherwise \end{cases}$$

for $k \in 0 \dots (N-1)$
where N is the length of $\bar{B}$

Parameters

- **a** – **[out]** Output BFP vector $\bar{A}$
- **b** – **[in]** Input BFP vector $\bar{B}$
- **lower_bound** – **[in]** Mantissa of the lower clipping bound, L
- **upper_bound** – **[in]** Mantissa of the upper clipping bound, U
- **bound_exp** – **[in]** Shared exponent of the clipping bounds

void **bfp_s16_rect**(*bfp_s16_t* *a, const *bfp_s16_t* *b)

Rectify a 16-bit BFP vector.

Each element A_k of output BFP vector $\bar{A}$ is set to the corresponding element B_k of input BFP vector $\bar{B}$ if it is non-negative, otherwise it is set to 0.

a and b must have been initialized (see *bfp_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$A_k \leftarrow \begin{cases} 0 & B_k < 0 \\ B_k & otherwise \end{cases}$$

for $k \in 0 \dots (N-1)$
where N is the length of $\bar{B}$

Parameters

- **a** – **[out]** Output BFP vector $\bar{A}$
- **b** – **[in]** Input BFP vector $\bar{B}$

void **bfp_s16_to_bfp_s32**(*bfp_s32_t* *a, const *bfp_s16_t* *b)

Convert a 16-bit BFP vector into a 32-bit BFP vector.

Increases the bit-depth of each 16-bit element B_k of input BFP vector $\bar{B}$ to 32 bits, and stores the 32-bit result in the corresponding element A_k of output BFP vector $\bar{A}$.

a and b must have been initialized (see *bfp_s16_init()* and *bfp_s32_init()*), and must be the same length.

Operation Performed

$$A_k \overset{32-bit}{\longleftarrow} B_k$$

for $k \in 0 \dots (N-1)$
where N is the length of $\bar{B}$

Parameters

- **a** − **[out]** Output BFP vector $\bar{A}$
- **b** − **[in]** Input BFP vector $\bar{B}$

void **bfp_s16_sqrt**(*bfp_s16_t* *a, const *bfp_s16_t* *b)

Get the square roots of elements of a 16-bit BFP vector.

Computes the square root of each element B_k of input BFP vector $\bar{B}$ and stores the results in output BFP vector $\bar{A}$.

a and b must have been initialized (see *bfp_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$A_k \leftarrow \sqrt{B_k}$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Notes

- Only the XMATH_BFP_SQRT_DEPTH_S16 (see xmath_conf.h) most significant bits of each result are computed.
- This function only computes real roots. For any $B_k < 0$, the corresponding output A_k is set to 0.

Parameters

- **a** − **[out]** Output BFP vector $\bar{A}$
- **b** − **[in]** Input BFP vector $\bar{B}$

void **bfp_s16_inverse**(*bfp_s16_t* *a, const *bfp_s16_t* *b)

Get the inverses of elements of a 16-bit BFP vector.

Computes the inverse of each element B_k of input BFP vector $\bar{B}$ and stores the results in output BFP vector $\bar{A}$.

a and b must have been initialized (see *bfp_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$A_k \leftarrow B_k^{-1}$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **a** − **[out]** Output BFP vector $\bar{A}$
- **b** − **[in]** Input BFP vector $\bar{B}$

float_s32_t **bfp_s16_abs_sum**(const *bfp_s16_t* *b)

Sum the absolute values of elements of a 16-bit BFP vector.

Sum the absolute values of elements of input BFP vector $\bar{B}$ for a result $A = a \cdot 2^{a\_exp}$, where a is a 32-bit mantissa and a_exp is its associated exponent. A is returned.

b must have been initialized (see *bfp_s16_init()*).

Operation Performed

$$A \leftarrow \sum_{k=0}^{N-1} |A_k|$$
where N is the length of $\bar{B}$

Parameters

- **b** − **[in]** Input BFP vector $\bar{B}$

Returns

A, the sum of absolute values of elements of $\bar{B}$

float **bfp_s16_mean**(const *bfp_s16_t* *b)

Get the mean value of a 16-bit BFP vector.

Computes $A = a \cdot 2^{a\_exp}$, the mean value of elements of input BFP vector $\bar{B}$, where a is the 16-bit mantissa of the result, and a_exp is its associated exponent. A is returned.

b must have been initialized (see *bfp_s16_init()*).

Operation Performed

$$A \leftarrow \frac{1}{N} \sum_{k=0}^{N-1} (B_k)$$
where N is the length of $\bar{B}$

Parameters

- **b** − **[in]** Input BFP vector $\bar{B}$

Returns

A, the mean value of $\bar{B}$'s elements

float_s64_t **bfp_s16_energy**(const *bfp_s16_t* *b)

Get the energy (sum of squared of elements) of a 16-bit BFP vector.

Computes $A = a \cdot 2^{a\_exp}$, the sum of squares of elements of input BFP vector $\bar{B}$, where a is the 64-bit mantissa of the result, and a_exp is its associated exponent. A is returned.

b must have been initialized (see *bfp_s16_init()*).

Operation Performed

$$A \leftarrow \sum_{k=0}^{N-1} (B_k^2)$$
where N is the length of $\bar{B}$

Parameters

- **b** – **[in]** Input BFP vector $\bar{B}$

Returns
> A, $\bar{B}$'s energy

float_s32_t **bfp_s16_rms**(const *bfp_s16_t* *b)

Get the RMS value of elements of a 16-bit BFP vector.

Computes $A = a \cdot 2^{a\text{-}exp}$, the RMS value of elements of input BFP vector $\bar{B}$, where a is the 32-bit mantissa of the result, and a_exp is its associated exponent. A is returned.

The RMS (root-mean-square) value of a vector is the square root of the sum of the squares of the vector's elements.

b must have been initialized (see *bfp_s16_init()*).

Operation Performed

$$A \leftarrow \sqrt{\frac{1}{N} \sum_{k=0}^{N-1} (B_k^2)}$$

where N is the length of $\bar{B}$

Parameters
- **b** – **[in]** Input BFP vector $\bar{B}$

Returns
> A, the RMS value of $\bar{B}$'s elements

float **bfp_s16_max**(const *bfp_s16_t* *b)

Get the maximum value of a 16-bit BFP vector.

Finds A, the maximum value among elements of input BFP vector $\bar{B}$. A is returned by this function.

b must have been initialized (see *bfp_s16_init()*).

Operation Performed

$$A \leftarrow max\left(B_0, B_1, ..., B_{N-1}\right)$$

where N is the length of $\bar{B}$

Parameters
- **b** – **[in]** Input vector

Returns
> A, the value of $\bar{B}$'s maximum element

void **bfp_s16_max_elementwise**(*bfp_s16_t* *a, const *bfp_s16_t* *b, const *bfp_s16_t* *c)

Get the element-wise maximum of two 16-bit BFP vectors.

Each element of output vector $\bar{A}$ is set to the maximum of the corresponding elements in the input vectors $\bar{B}$ and $\bar{C}$.

a, b and c must have been initialized (see *bfp_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b, but *not* on c.

Operation Performed

$$A_k \leftarrow max(B_k, C_k)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **a** – Output BFP vector $\bar{A}$
- **b** – Input BFP vector $\bar{B}$
- **c** – Input BFP vector $\bar{C}$

float **bfp_s16_min**(const *bfp_s16_t* *b)

Get the minimum value of a 16-bit BFP vector.

Finds A, the minimum value among elements of input BFP vector $\bar{B}$. A is returned by this function.

b must have been initialized (see *bfp_s16_init()*).

Operation Performed

$$A \leftarrow min\left(B_0, B_1, \dots, B_{N-1}\right)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **b** – **[in]** Input vector

Returns

A, the value of $\bar{B}$'s minimum element

void **bfp_s16_min_elementwise**(*bfp_s16_t* *a, const *bfp_s16_t* *b, const *bfp_s16_t* *c)

Get the element-wise minimum of two 16-bit BFP vectors.

Each element of output vector $\bar{A}$ is set to the minimum of the corresponding elements in the input vectors $\bar{B}$ and $\bar{C}$.

a, b and c must have been initialized (see *bfp_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b, but *not* on c.

Operation Performed

$$A_k \leftarrow min(B_k, C_k)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **a** – Output BFP vector $\bar{A}$
- **b** – Input BFP vector $\bar{B}$
- **c** – Input BFP vector $\bar{C}$

unsigned **bfp_s16_argmax**(const *bfp_s16_t* *b)

Get the index of the maximum value of a 16-bit BFP vector.

Finds a, the index of the maximum value among the elements of input BFP vector $\bar{B}$. a is returned by this function.

If `i` is the value returned, then the maximum value in $\bar{B}$ is `ldexp(b->data[i],b->exp)`.

Operation Performed

$$a \leftarrow argmax_k\,(b_k)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Notes

- If there is a tie for maximum value, the lowest tying index is returned.

Parameters

- **b** − **[in]** Input vector

Returns

a, the index of the maximum value from $\bar{B}$

unsigned **bfp_s16_argmin**(const *bfp_s16_t* *b)

Get the index of the minimum value of a 16-bit BFP vector.

Finds a, the index of the minimum value among the elements of input BFP vector $\bar{B}$. a is returned by this function.

If `i` is the value returned then the minimum value in $\bar{B}$ is `ldexp(b->data[i], b->exp)`.

Operation Performed

$$a \leftarrow argmin_k\,(b_k)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Notes

- If there is a tie for minimum value, the lowest tying index is returned.

Parameters

- **b** − **[in]** Input vector

Returns

a, the index of the minimum value from $\bar{B}$

headroom_t **bfp_s16_accumulate**(*split_acc_s32_t* a[], const *exponent_t* a_exp, const *bfp_s16_t* *b)

Accumulate a 16-bit BFP vector into a 32-bit accumulator vector.

This function is used for efficiently accumulating a series of 16-bit BFP vectors into a 32-bit vector. Each call to this function adds a BFP vector $\bar{B}$ into the persistent 32-bit accumulator vector $\bar{A}$.

Eventually the value of $\bar{A}$ will be needed for something other than simple accumulation, which requires converting from the XS3-native split accumulator representation given by the

split_acc_s32_t struct, into a standard vector of `int32_t`. This can be accomplished using *vect_s32_merge_accs()*. From there, the `int32_t` vector can be dropped to a 16-bit vector with *vect_s32_to_vect_s16()* if needed.

Note, in order for this operation to work, `b_exp` − `a_exp` must be no greater than 14.

Operation Performed

$$\bar{A} \leftarrow \bar{A} + \bar{B}$$

Proper use of this function requires some book-keeping on the part of the caller. In particular, the caller is responsible for tracking the exponent and monitoring the headroom of the accumulator vector $\bar{A}$.

Usage

To begin a sequence of accumulation, start by clearing the contents of $\bar{A}$ to all zeros. Then, an appropriate exponent for $\bar{A}$ must be chosen. The only hard constraint is that the accumulator exponent, `a_exp` must be within 14 of $\bar{B}$'s exponent, `b_exp`. If `b_exp` is unknown, the caller may choose to wait until the first $\bar{B}$ is available before initializing `a_exp`.

As vectors are accumulated into $\bar{A}$ with multiple calls to this function, it becomes possible for $\bar{A}$ to saturate for some element. Each call to this function returns the headroom of $\bar{A}$ (note: no more than 15 bits of headroom will be reported). If $\bar{A}$ has at least 1 bit of headroom, then a call to this function is guarranteed not to saturate.

The larger `a_exp` is compared to each `b_exp`, the more 16-bit vectors can be accumulated before saturation becomes possible (and by virtue of that, the more efficiently accumulation can take place.). On the other hand, as long as `a_exp` $\leq$ `b_exp`, there is no precision loss during accumulation. It is the responsibility of the caller to manage this trade-off.

If and when this function reports that $\bar{A}$ has 0 headroom, if further accumulation is needed, the caller can handle this by increasing `a_exp`. Increasing `a_exp` will require that the contents of the mantissa vector $\bar{a}$ be right-shifted to avoid corrupting the value of $\bar{A}$, making room for further accumulation in the process. Shifting the split accumulators can be accomplished with a call to *vect_split_acc_s32_shr()*.

Finally, when accumulation is complete or the accumulator values must be used elsewhere, the split accumulator vector can be converted to simple `int32_t` vector with a call to *vect_s32_merge_accs()*.

Parameters

- **a** − **[inout]** Mantissas of accumulator vector $\bar{A}$
- **a_exp** − **[in]** Exponent of accumulator vector $\bar{A}$
- **b** − **[in]** Input vector $\bar{B}$

Returns

Headroom of $\bar{A}$ (up to 15 bits)

4.2.3 32-Bit Block Floating-Point API

group **bfp_s32_api**

Functions

void **bfp_s32_init**(*bfp_s32_t* *a, int32_t *data, const *exponent_t* exp, const unsigned length, const unsigned calc_hr)

Initialize a 32-bit BFP vector.

This function initializes each of the fields of BFP vector `a`.

`data` points to the memory buffer used to store elements of the vector, so it must be at least `length * 4` bytes long, and must begin at a word-aligned address.

`exp` is the exponent assigned to the BFP vector. The logical value associated with the `k`th element of the vector after initialization is $data_k \cdot 2^{exp}$.

If `calc_hr` is false, `a->hr` is initialized to 0. Otherwise, the headroom of the the BFP vector is calculated and used to initialize `a->hr`.

Parameters

- **a** – **[out]** BFP vector to initialize
- **data** – **[in]** `int32_t` buffer used to back `a`
- **exp** – **[in]** Exponent of BFP vector
- **length** – **[in]** Number of elements in the BFP vector
- **calc_hr** – **[in]** Boolean indicating whether the HR of the BFP vector should be calculated

bfp_s32_t **bfp_s32_alloc**(unsigned length)

Dynamically allocate a 32-bit BFP vector from the heap.

If allocation was unsuccessful, the `data` field of the returned vector will be NULL, and the `length` field will be zero. Otherwise, `data` will point to the allocated memory and the `length` field will be the user-specified length. The `length` argument must not be zero.

Neither the BFP exponent, headroom, nor the elements of the allocated mantissa vector are set by this function. To set the BFP vector elements to a known value, use *bfp_s32_set()* on the retuned BFP vector.

BFP vectors allocated using this function must be deallocated using *bfp_s32_dealloc()* to avoid a memory leak.

To initialize a BFP vector using static memory allocation, use *bfp_s32_init()* instead.

See also:

bfp_s32_dealloc, *bfp_s32_init*

Note: This function always allocates an extra 2 elements so that *bfp_fft_unpack_mono()* can safely be used, but these two elements will NOT be reflected in the returned vector length.

Parameters

- **length** – **[in]** The length of the BFP vector to be allocated (in elements)

Returns
32-bit BFP vector

void **bfp_s32_dealloc**(*bfp_s32_t* *vector)

Deallocate a 32-bit BFP vector allocated by *bfp_s32_alloc()*.

Use this function to free the heap memory allocated by *bfp_s32_alloc()*.

BFP vectors whose mantissa buffer was (successfully) dynamically allocated have a flag set which indicates as much. This function can safely be called on any *bfp_s32_t* which has not had its `flags` or `data` manually manipulated, including:

- *bfp_s32_t* resulting from a successful call to *bfp_s32_alloc()*
- *bfp_s32_t* resulting from an unsuccessful call to *bfp_s32_alloc()*
- *bfp_s32_t* initialized with a call to *bfp_s32_init()*

In the latter two cases, this function does nothing. In the former, the `data`, `length` and `flags` fields of `vector` are cleared to zero.

See also:

bfp_s32_alloc

Parameters

- **vector** – **[in]** BFP vector to be deallocated.

void **bfp_s32_set**(*bfp_s32_t* *a, const int32_t b, const *exponent_t* exp)

Set all elements of a 32-bit BFP vector to a specified value.

The exponent of a is set to `exp`, and each element's mantissa is set to `b`.

After performing this operation, all elements will represent the same value $b \cdot 2^{exp}$.

a must have been initialized (see *bfp_s32_init()*).

Parameters

- **a** – **[out]** BFP vector to update
- **b** – **[in]** New value each mantissa is set to
- **exp** – **[in]** New exponent for the BFP vector

void **bfp_s32_use_exponent**(*bfp_s32_t* *a, const *exponent_t* exp)

Modify a 32-bit BFP vector to use a specified exponent.

This function forces BFP vector $\bar{A}$ to use a specified exponent. The mantissa vector $\bar{a}$ will be bit-shifted left or right to compensate for the changed exponent.

This function can be used, for example, before calling a fixed-point arithmetic function to ensure the underlying mantissa vector has the needed Q-format. As another example, this may be useful when communicating with peripheral devices (e.g. via I2S) that require sample data to be in a specified format.

Note that this sets the *current* encoding, and does not *fix* the exponent permanently (i.e. subsequent operations may change the exponent as usual).

If the required fixed-point Q-format is `QX.Y`, where `Y` is the number of fractional bits in the resulting mantissas, then the associated exponent (and value for parameter `exp`) is `-Y`.

`a` points to input BFP vector $\bar{A}$, with mantissa vector $\bar{a}$ and exponent a_exp. `a` is updated in place to produce resulting BFP vector $\tilde{A}$ with mantissa vector $\tilde{a}$ and exponent $\tilde{a}\_exp$.

`exp` is $\tilde{a}\_exp$, the required exponent. $\Delta p = \tilde{a}\_exp - a\_exp$ is the required change in exponent.

If $\Delta p = 0$, the BFP vector is left unmodified.

If $\Delta p > 0$, the required exponent is larger than the current exponent and an arithmetic right-shift of Δp bits is applied to the mantissas $\bar{a}$. When applying a right-shift, precision may be lost by discarding the Δp least significant bits.

If $\Delta p < 0$, the required exponent is smaller than the current exponent and a left-shift of Δp bits is applied to the mantissas $\bar{a}$. When left-shifting, saturation logic will be applied such that any element that can't be represented exactly with the new exponent will saturate to the 32-bit saturation bounds.

The exponent and headroom of `a` are updated by this function.

Operation Performed

$$\Delta p = \tilde{a}\_exp - a\_exp$$
$$\tilde{a}_k \leftarrow sat_{32}(a_k \cdot 2^{-\Delta p})$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{A} \text{ (in elements)}$$

Parameters

- **a** – **[inout]** Input BFP vector $\bar{A}$ / Output BFP vector $\tilde{A}$
- **exp** – **[in]** The required exponent, $\tilde{a}\_exp$

headroom_t **bfp_s32_headroom**(*bfp_s32_t* *b)

Get the headroom of a 32-bit BFP vector.

The headroom of a vector is the number of bits its elements can be left-shifted without losing any information. It conveys information about the range of values that vector may contain, which is useful for determining how best to preserve precision in potentially lossy block floating-point operations.

In a BFP context, headroom applies to mantissas only, not exponents.

In particular, if the 32-bit mantissa vector $\bar{x}$ has N bits of headroom, then for any element x_k of $\bar{x}$

$$-2^{31-N} \le x_k < 2^{31-N}$$

And for any element $X_k = x_k \cdot 2^{x\_exp}$ of a complex BFP vector $\bar{X}$

$$-2^{31+x\_exp-N} \le X_k < 2^{31+x\_exp-N}$$

This function determines the headroom of `b`, updates `b->hr` with that value, and then returns `b->hr`.

Parameters

- **b** – BFP vector to get the headroom of

Returns

Headroom of BFP vector `b`

void **bfp_s32_shl**(*bfp_s32_t* *a, const *bfp_s32_t* *b, const *left_shift_t* b_shl)

Apply a left-shift to the mantissas of a 32-bit BFP vector.

Each mantissa of input BFP vector $\bar{B}$ is left-shifted b_shl bits and stored in the corresponding element of output BFP vector $\bar{A}$.

This operation can be used to add or remove headroom from a BFP vector.

b_shl is the number of bits that each mantissa will be left-shifted. This shift is signed and arithmetic, so negative values for b_shl will right-shift the mantissas.

a and b must have been initialized (see *bfp_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Note that this operation bypasses the logic protecting the caller from saturation or underflows. Output values saturate to the symmetric 32-bit range (the open interval $(-2^{31}, 2^{31})$). To avoid saturation, b_shl should be no greater than the headroom of b (b->hr).

Operation Performed

$$a_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{b\_shl} \rfloor)$$
$$\text{for } k \in 0 \ldots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$
$$\text{and } b_k \text{ and } a_k \text{ are the } k\text{th mantissas from } \bar{B} \text{ and } \bar{A} \text{ respectively}$$

Parameters

- **a** − **[out]** Output BFP vector $\bar{A}$
- **b** − **[in]** Input BFP vector $\bar{B}$
- **b_shl** − **[in]** Signed arithmetic left-shift to be applied to mantissas of $\bar{B}$.

void **bfp_s32_add**(*bfp_s32_t* *a, const *bfp_s32_t* *b, const *bfp_s32_t* *c)

Add two 32-bit BFP vectors together.

Add together two input BFP vectors $\bar{B}$ and $\bar{C}$ and store the result in BFP vector $\bar{A}$.

a, b and c must have been initialized (see *bfp_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b or c.

Operation Performed

$$\bar{A} \leftarrow \bar{B} + \bar{C}$$

Parameters

- **a** − **[out]** Output BFP vector $\bar{A}$
- **b** − **[in]** Input BFP vector $\bar{B}$
- **c** − **[in]** Input BFP vector $\bar{C}$

void **bfp_s32_add_scalar**(*bfp_s32_t* *a, const *bfp_s32_t* *b, const *float_s32_t* c)

Add a scalar to a 32-bit BFP vector.

Add a real scalar c to input BFP vector $\bar{B}$ and store the result in BFP vector $\bar{A}$.

a, and b must have been initialized (see *bfp_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$\bar{A} \leftarrow \bar{B} + c$$

Parameters

- **a** – **[out]** Output BFP vector $\bar{A}$
- **b** – **[in]** Input BFP vector $\bar{B}$
- **c** – **[in]** Input scalar c

void **bfp_s32_sub**(*bfp_s32_t* *a, const *bfp_s32_t* *b, const *bfp_s32_t* *c)

Subtract one 32-bit BFP vector from another.

Subtract input BFP vector $\bar{C}$ from input BFP vector $\bar{C}$ and store the result in BFP vector $\bar{A}$.

a, b and c must have been initialized (see *bfp_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b or c.

Operation Performed

$$\bar{A} \leftarrow \bar{B} - \bar{C}$$

Parameters

- **a** – **[out]** Output BFP vector $\bar{A}$
- **b** – **[in]** Input BFP vector $\bar{B}$
- **c** – **[in]** Input BFP vector $\bar{C}$

void **bfp_s32_mul**(*bfp_s32_t* *a, const *bfp_s32_t* *b, const *bfp_s32_t* *c)

Multiply one 32-bit BFP vector by another element-wise.

Multiply each element of input BFP vector $\bar{B}$ by the corresponding element of input BFP vector $\bar{C}$ and store the results in output BFP vector $\bar{A}$.

a, b and c must have been initialized (see *bfp_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b or c.

Operation Performed

$$A_k \leftarrow B_k \cdot C_k$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **a** – Output BFP vector $\bar{A}$
- **b** – Input BFP vector $\bar{B}$
- **c** – Input BFP vector $\bar{C}$

void **bfp_s32_macc**(*bfp_s32_t* *acc, const *bfp_s32_t* *b, const *bfp_s32_t* *c)

Multiply one 32-bit BFP vector by another element-wise and add the result to a third vector.

Operation Performed

$$A_k \leftarrow A_k + B_k \cdot C_k$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- `acc` − **[inout]** Input/Output accumulator BFP vector $\bar{A}$
- `b` − **[in]** Input BFP vector $\bar{B}$
- `c` − **[in]** Input BFP vector $\bar{C}$

void **bfp_s32_nmacc**(*bfp_s32_t* *acc, const *bfp_s32_t* *b, const *bfp_s32_t* *c)

Multiply one 32-bit BFP vector by another element-wise and subtract the result from a third vector.

Operation Performed

$$A_k \leftarrow A_k - B_k \cdot C_k$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- `acc` − **[inout]** Input/Output accumulator BFP vector $\bar{A}$
- `b` − **[in]** Input BFP vector $\bar{B}$
- `c` − **[in]** Input BFP vector $\bar{C}$

void **bfp_s32_scale**(*bfp_s32_t* *a, const *bfp_s32_t* *b, const *float_s32_t* alpha)

Multiply a 32-bit BFP vector by a scalar.

Multiply input BFP vector $\bar{B}$ by scalar $\alpha \cdot 2^{\alpha\_exp}$ and store the result in output BFP vector $\bar{A}$.

a and b must have been initialized (see *bfp_s32_init()*), and must be the same length.

alpha represents the scalar $\alpha \cdot 2^{\alpha\_exp}$, where α is `alpha.mant` and α_exp is `alpha.exp`.

This operation can be performed safely in-place on b.

Operation Performed

$$\bar{A} \leftarrow \bar{B} \cdot (\alpha \cdot 2^{\alpha\_exp})$$

Parameters

- `a` − **[out]** Output BFP vector $\bar{A}$
- `b` − **[in]** Input BFP vector $\bar{B}$
- `alpha` − **[in]** Scalar by which $\bar{B}$ is multiplied

void **bfp_s32_abs**(*bfp_s32_t* *a, const *bfp_s32_t* *b)

Get the absolute values of elements of a 32-bit BFP vector.

Compute the absolute value of each element B_k of input BFP vector $\bar{B}$ and store the results in output BFP vector $\bar{A}$.

a and b must have been initialized (see *bfp_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b.

$$A_k \leftarrow |B_k|$$
$$\text{for } k \in 0 \ldots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **a** – **[out]** Output BFP vector $\bar{A}$
- **b** – **[in]** Input BFP vector $\bar{B}$

float_s64_t **bfp_s32_sum**(const *bfp_s32_t* *b)

Sum the elements of a 32-bit BFP vector.

Sum the elements of input BFP vector $\bar{B}$ to get a result $A = a \cdot 2^{a\_exp}$, which is returned. The returned value has a 64-bit mantissa.

b must have been initialized (see *bfp_s32_init()*).

Operation Performed

$$A \leftarrow \sum_{k=0}^{N-1} (B_k)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **b** – **[in]** Input BFP vector $\bar{B}$

Returns

A, the sum of elements of $\bar{B}$

float_s64_t **bfp_s32_dot**(const *bfp_s32_t* *b, const *bfp_s32_t* *c)

Compute the inner product of two 32-bit BFP vectors.

Adds together the element-wise products of input BFP vectors $\bar{B}$ and $\bar{C}$ for a result $A = a \cdot 2^{a\_exp}$, where a is the 64-bit mantissa of the result and a_exp is its associated exponent. A is returned.

b and c must have been initialized (see *bfp_s32_init()*), and must be the same length.

Operation Performed

$$a \cdot 2^{a\_exp} \leftarrow \sum_{k=0}^{N-1} (B_k \cdot C_k)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **b** – **[in]** Input BFP vector $\bar{B}$
- **c** – **[in]** Input BFP vector $\bar{C}$

Returns

A, the inner product of vectors $\bar{B}$ and $\bar{C}$

void **bfp_s32_clip**(*bfp_s32_t* *a, const *bfp_s32_t* *b, const int32_t lower_bound, const int32_t upper_bound, const int bound_exp)

    Clamp the elements of a 32-bit BFP vector to a specified range.

    Each element A_k of output BFP vector $\bar{A}$ is set to the corresponding element B_k of input BFP vector $\bar{B}$ if it is in the range $[L \cdot 2^{bound\_exp}, U \cdot 2^{bound\_exp}]$, otherwise it is set to the nearest value inside that range.

    a and b must have been initialized (see *bfp_s32_init()*), and must be the same length.

    This operation can be performed safely in-place on b.

    Operation Performed

$$A_k \leftarrow \begin{cases} L \cdot 2^{bound\_exp} & B_k < L \cdot 2^{bound\_exp} \\ U \cdot 2^{bound\_exp} & B_k > U \cdot 2^{bound\_exp} \\ B_k & otherwise \end{cases}$$

$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

        Parameters

- **a** – **[out]** Output BFP vector $\bar{A}$
- **b** – **[in]** Input BFP vector $\bar{B}$
- **lower_bound** – **[in]** Mantissa of the lower clipping bound, L
- **upper_bound** – **[in]** Mantissa of the upper clipping bound, U
- **bound_exp** – **[in]** Shared exponent of the clipping bounds

void **bfp_s32_rect**(*bfp_s32_t* *a, const *bfp_s32_t* *b)

    Rectify a 32-bit BFP vector.

    Each element A_k of output BFP vector $\bar{A}$ is set to the corresponding element B_k of input BFP vector $\bar{B}$ if it is non-negative, otherwise it is set to 0.

    a and b must have been initialized (see *bfp_s32_init()*), and must be the same length.

    This operation can be performed safely in-place on b.

    Operation Performed

$$A_k \leftarrow \begin{cases} 0 & B_k < 0 \\ B_k & otherwise \end{cases}$$

$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

        Parameters

- **a** – **[out]** Output BFP vector $\bar{A}$
- **b** – **[in]** Input BFP vector $\bar{B}$

void **bfp_s32_to_bfp_s16**(*bfp_s16_t* *a, const *bfp_s32_t* *b)

    Convert a 32-bit BFP vector into a 16-bit BFP vector.

    Reduces the bit-depth of each 32-bit element B_k of input BFP vector $\bar{B}$ to 16 bits, and stores the 16-bit result in the corresponding element A_k of output BFP vector $\bar{A}$.

a and b must have been initialized (see *bfp_s32_init()* and *bfp_s16_init()*), and must be the same length.

As much precision as possible will be retained.

Operation Performed

$$A_k \overset{16-bit}{\longleftarrow} B_k$$
$$\text{for } k \in 0 \ldots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **a** – **[out]** Output BFP vector $\bar{A}$
- **b** – **[in]** Input BFP vector $\bar{B}$

void **bfp_s32_sqrt**(*bfp_s32_t* *a, const *bfp_s32_t* *b)

Get the square roots of elements of a 32-bit BFP vector.

Computes the square root of each element B_k of input BFP vector $\bar{B}$ and stores the results in output BFP vector $\bar{A}$.

a and b must have been initialized (see *bfp_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$A_k \leftarrow \sqrt{B_k}$$
$$\text{for } k \in 0 \ldots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Notes

- Only the XMATH_BFP_SQRT_DEPTH_S32 (see xmath_conf.h) most significant bits of each result are computed.
- This function only computes real roots. For any $B_k < 0$, the corresponding output A_k is set to 0.

Parameters

- **a** – **[out]** Output BFP vector $\bar{A}$
- **b** – **[in]** Input BFP vector $\bar{B}$

void **bfp_s32_inverse**(*bfp_s32_t* *a, const *bfp_s32_t* *b)

Get the inverses of elements of a 32-bit BFP vector.

Computes the inverse of each element B_k of input BFP vector $\bar{B}$ and stores the results in output BFP vector $\bar{A}$.

a and b must have been initialized (see *bfp_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$A_k \leftarrow B_k^{-1}$$
$$\text{for } k \in 0 \ldots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **a** − **[out]** Output BFP vector $\bar{A}$
- **b** − **[in]** Input BFP vector $\bar{B}$

float_s64_t **bfp_s32_abs_sum**(const *bfp_s32_t* *b)

Sum the absolute values of elements of a 32-bit BFP vector.

Sum the absolute values of elements of input BFP vector $\bar{B}$ for a result $A = a \cdot 2^{a\_exp}$, where a is a 64-bit mantissa and a_exp is its associated exponent. A is returned.

b must have been initialized (see *bfp_s32_init()*).

Operation Performed

$$A \leftarrow \sum_{k=0}^{N-1} |A_k|$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **b** − **[in]** Input BFP vector $\bar{B}$

Returns

A, the sum of absolute values of elements of $\bar{B}$

float_s32_t **bfp_s32_mean**(const *bfp_s32_t* *b)

Get the mean value of a 32-bit BFP vector.

Computes $A = a \cdot 2^{a\_exp}$, the mean value of elements of input BFP vector $\bar{B}$, where a is the 32-bit mantissa of the result, and a_exp is its associated exponent. A is returned.

b must have been initialized (see *bfp_s32_init()*).

Operation Performed

$$A \leftarrow \frac{1}{N} \sum_{k=0}^{N-1} (B_k)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **b** − **[in]** Input BFP vector $\bar{B}$

Returns

A, the mean value of $\bar{B}$'s elements

float_s64_t **bfp_s32_energy**(const *bfp_s32_t* *b)

Get the energy (sum of squared of elements) of a 32-bit BFP vector.

Computes $A = a \cdot 2^{a\_exp}$, the sum of squares of elements of input BFP vector $\bar{B}$, where a is the 64-bit mantissa of the result, and a_exp is its associated exponent. A is returned.

b must have been initialized (see *bfp_s32_init()*).

Operation Performed

$$A \leftarrow \sum_{k=0}^{N-1} \left(B_k^2 \right)$$

where N is the length of $\bar{B}$

Parameters

- **b** − **[in]** Input BFP vector $\bar{B}$

Returns

A, $\bar{B}$'s energy

float_s32_t **bfp_s32_rms**(const *bfp_s32_t* *b)

Get the RMS value of elements of a 32-bit BFP vector.

Computes $A = a \cdot 2^{a\_exp}$, the RMS value of elements of input BFP vector $\bar{B}$, where a is the 32-bit mantissa of the result, and a_exp is its associated exponent. A is returned.

The RMS (root-mean-square) value of a vector is the square root of the sum of the squares of the vector's elements.

b must have been initialized (see *bfp_s32_init()*).

Operation Performed

$$A \leftarrow \sqrt{\frac{1}{N} \sum_{k=0}^{N-1} \left(B_k^2 \right)}$$

where N is the length of $\bar{B}$

Parameters

- **b** − **[in]** Input BFP vector $\bar{B}$

Returns

A, the RMS value of $\bar{B}$'s elements

float_s32_t **bfp_s32_max**(const *bfp_s32_t* *b)

Get the maximum value of a 32-bit BFP vector.

Finds A, the maximum value among elements of input BFP vector $\bar{B}$. A is returned by this function.

b must have been initialized (see *bfp_s32_init()*).

Operation Performed

$$A \leftarrow max \left(B_0, B_1, ..., B_{N-1} \right)$$

where N is the length of $\bar{B}$

Parameters

- **b** − **[in]** Input vector

Returns
> A, the value of $\bar{B}$'s maximum element

void **bfp_s32_max_elementwise**(*bfp_s32_t* *a, const *bfp_s32_t* *b, const *bfp_s32_t* *c)

Get the element-wise maximum of two 32-bit BFP vectors.

Each element of output vector $\bar{A}$ is set to the maximum of the corresponding elements in the input vectors $\bar{B}$ and $\bar{C}$.

a, b and c must have been initialized (see *bfp_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b, but *not* on c.

Operation Performed

$$A_k \leftarrow max(B_k, C_k)$$
$$\text{for } k \in 0 \text{ ... } (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **a** − Output BFP vector $\bar{A}$
- **b** − Input BFP vector $\bar{B}$
- **c** − Input BFP vector $\bar{C}$

float_s32_t **bfp_s32_min**(const *bfp_s32_t* *b)

Get the minimum value of a 32-bit BFP vector.

Finds A, the minimum value among elements of input BFP vector $\bar{B}$. A is returned by this function.

b must have been initialized (see *bfp_s32_init()*).

Operation Performed

$$A \leftarrow min\left(B_0, B_1, ..., B_{N-1}\right)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **b** − **[in]** Input vector

Returns
> A, the value of $\bar{B}$'s minimum element

void **bfp_s32_min_elementwise**(*bfp_s32_t* *a, const *bfp_s32_t* *b, const *bfp_s32_t* *c)

Get the element-wise minimum of two 32-bit BFP vectors.

Each element of output vector $\bar{A}$ is set to the minimum of the corresponding elements in the input vectors $\bar{B}$ and $\bar{C}$.

a, b and c must have been initialized (see *bfp_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b, but *not* on c.

Operation Performed

$$A_k \leftarrow min(B_k, C_k)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **a** − Output BFP vector $\bar{A}$
- **b** − Input BFP vector $\bar{B}$
- **c** − Input BFP vector $\bar{C}$

unsigned **bfp_s32_argmax**(const *bfp_s32_t* *b)

Get the index of the maximum value of a 32-bit BFP vector.

Finds a, the index of the maximum value among the elements of input BFP vector $\bar{B}$. a is returned by this function.

If `i` is the value returned, then the maximum value in $\bar{B}$ is `ldexp(b->data[i], b->exp)`.

Operation Performed

$$a \leftarrow argmax_k (b_k)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Notes

- If there is a tie for maximum value, the lowest tying index is returned.

Parameters

- **b** − **[in]** Input vector

Returns

a, the index of the maximum value from $\bar{B}$

unsigned **bfp_s32_argmin**(const *bfp_s32_t* *b)

Get the index of the minimum value of a 32-bit BFP vector.

Finds a, the index of the minimum value among the elements of input BFP vector $\bar{B}$. a is returned by this function.

If `i` is the value returned, then the minimum value in $\bar{B}$ is `ldexp(b->data[i], b->exp)`.

Operation Performed

$$a \leftarrow argmin_k (b_k)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Notes

- If there is a tie for minimum value, the lowest tying index is returned.

Parameters

- **b** − **[in]** Input vector

Returns

a, the index of the minimum value from $\bar{B}$

void **bfp_s32_convolve_valid**(*bfp_s32_t* *y, const *bfp_s32_t* *x, const int32_t b_q30[], const unsigned b_length)

Convolve a 32-bit BFP vector with a short convolution kernel ("valid" mode).

Input BFP vector $\bar{X}$ is convolved with a short fixed-point convolution kernel $\bar{b}$ to produce output BFP vector $\bar{Y}$. In other words, this function applies the Kth-order FIR filter with coefficients given by $\bar{b}$ to the input signal $\bar{X}$. The convolution is "valid" in the sense that no output elements are emitted where the filter taps extend beyond the bounds of the input vector, resulting in an output vector $\bar{Y}$ with fewer elements.

The maximum filter order K supported by this function is 7.

y is the output vector $\bar{Y}$. If input $\bar{X}$ has N elements, and the filter has K coefficients, then $\bar{Y}$ has $N - 2P$ elements, where $P = \lfloor K/2 \rfloor$.

x is the input vector $\bar{X}$ with length N and elements.

b_q30[] is the vector $\bar{b}$ of filter coefficients. The coefficients of $\bar{b}$ are encoded in a Q2.30 fixed-point format. The effective value of the ith coefficient is then $b_i \cdot 2^{-30}$.

b_length is the length K of $\bar{b}$ in elements (i.e. the number of filter taps). b_length must be one of $\{1, 3, 5, 7\}$.

Operation Performed

$$Y_k \leftarrow \sum_{l=0}^{K-1} (X_{(k+l)} \cdot b_l \cdot 2^{-30})$$
$$\text{for } k \in 0 \ldots (N - 2P)$$
$$\text{where } P = \lfloor K/2 \rfloor$$

Parameters

- **y** − **[out]** Output BFP vector $\bar{Y}$
- **x** − **[in]** Input BFP vector $\bar{X}$
- **b_q30** − **[in]** Convolution kernel $\bar{b}$
- **b_length** − **[in]** The number of elements K in $\bar{b}$

void **bfp_s32_convolve_same**(*bfp_s32_t* *y, const *bfp_s32_t* *x, const int32_t b_q30[], const unsigned b_length, const *pad_mode_e* padding_mode)

Convolve a 32-bit BFP vector with a short convolution kernel ("same" mode).

Input BFP vector $\bar{X}$ is convolved with a short fixed-point convolution kernel $\bar{b}$ to produce output BFP vector $\bar{Y}$. In other words, this function applies the Kth-order FIR filter with coefficients given by $\bar{b}$ to the input signal $\bar{X}$. The convolution mode is "same" in that the input vector is effectively padded such that the input and output vectors are the same length. The padding behavior is one of those given by *pad_mode_e*.

The maximum filter order K supported by this function is 7.

y and x are the output and input BFP vectors $\bar{Y}$ and $\bar{X}$ respectively.

b_q30[] is the vector $\bar{b}$ of filter coefficients. The coefficients of $\bar{b}$ are encoded in a Q2.30 fixed-point format. The effective value of the ith coefficient is then $b_i \cdot 2^{-30}$.

b_length is the length K of $\bar{b}$ in elements (i.e. the number of filter taps). b_length must be one of $\{1, 3, 5, 7\}$.

padding_mode is one of the values from the *pad_mode_e* enumeration. The padding mode indicates the filter input values for filter taps that have extended beyond the bounds of the input vector $\bar{X}$. See *pad_mode_e* for a list of supported padding modes and associated behaviors.

Operation Performed

$$\tilde{x}_i = \begin{cases} \text{determined by padding mode} & i < 0 \\ \text{determined by padding mode} & i \geq N \\ x_i & otherwise \end{cases}$$

$$y_k \leftarrow \sum_{l=0}^{K-1} (\tilde{x}_{(k+l-P)} \cdot b_l \cdot 2^{-30})$$

$$\text{for } k \in 0 \ldots (N - 2P)$$

$$\text{where } P = \lfloor K/2 \rfloor$$

Note: Unlike *bfp_s32_convolve_valid()*, this operation *cannot* be performed safely in-place on x

---

Parameters

- **y** − **[out]** Output BFP vector $\bar{Y}$
- **x** − **[in]** Input BFP vector $\bar{X}$
- **b_q30** − **[in]** Convolution kernel $\bar{b}$
- **b_length** − **[in]** The number of elements K in $\bar{b}$
- **padding_mode** − **[in]** The padding mode to be applied at signal boundaries

4.2.4 Complex 16-Bit Block Floating-Point API

group **bfp_complex_s16_api**

Functions

void **bfp_complex_s16_init**(*bfp_complex_s16_t* *a, int16_t *real_data, int16_t *imag_data, const *exponent_t* exp, const unsigned length, const unsigned calc_hr)

Initialize a complex 16-bit BFP vector.

This function initializes each of the fields of BFP vector a.

Unlike complex 32-bit BFP vectors (*bfp_complex_s16_t*), for the sake of various optimizations the real and imaginary parts of elements' mantissas are stored in separate memory buffers.

real_data points to the memory buffer used to store the real part of each mantissa. It must be at least length * 2 bytes long, and must begin at a word-aligned address.

imag_data points to the memory buffer used to store the imaginary part of each mantissa. It must be at least length * 2 bytes long, and must begin at a word-aligned address.

exp is the exponent assigned to the BFP vector. The logical value associated with the kth element of the vector after initialization is $data_k \cdot 2^{exp}$.

If `calc_hr` is false, `a->hr` is initialized to 0. Otherwise, the headroom of the the BFP vector is calculated and used to initialize `a->hr`.

Parameters

- **a** – **[out]** BFP vector to initialize
- **real_data** – **[in]** `int16_t` buffer used to back the real part of `a`
- **imag_data** – **[in]** `int16_t` buffer used to back the imaginary part of `a`
- **exp** – **[in]** Exponent of BFP vector
- **length** – **[in]** Number of elements in BFP vector
- **calc_hr** – **[in]** Boolean indicating whether the HR of the BFP vector should be calculated

bfp_complex_s16_t **bfp_complex_s16_alloc**(const unsigned length)

Dynamically allocate a complex 16-bit BFP vector from the heap.

If allocation was unsuccessful, the `real` and `imag` fields of the returned vector will be NULL, and the `length` field will be zero. Otherwise, `real` and `imag` will point to the allocated memory and the `length` field will be the user-specified length. The `length` argument must not be zero.

This function allocates a single block of memory for both the real and imaginary parts of the BFP vector. Because all BFP functions require the mantissa buffers to begin at a word- aligned address, if `length` is odd, this function will allocate an extra `int16_t` element for the buffer.

Neither the BFP exponent, headroom, nor the elements of the allocated mantissa vector are set by this function. To set the BFP vector elements to a known value, use *bfp_complex_s16_set()* on the retuned BFP vector.

BFP vectors allocated using this function must be deallocated using *bfp_complex_s16_dealloc()* to avoid a memory leak.

To initialize a BFP vector using static memory allocation, use *bfp_complex_s16_init()* instead.

See also:

bfp_complex_s16_dealloc, *bfp_complex_s16_init*

Note: Dynamic allocation of BFP vectors relies on allocation from the heap, and offers no guarantees about the execution time. Use of this function in any time-critical section of code is highly discouraged.

Parameters

- **length** – **[in]** The length of the BFP vector to be allocated (in elements)

Returns

Complex 16-bit BFP vector

void **bfp_complex_s16_dealloc**(*bfp_complex_s16_t* *vector)

Deallocate a complex 16-bit BFP vector allocated by *bfp_complex_s16_alloc()*.

Use this function to free the heap memory allocated by *bfp_complex_s16_alloc()*.

BFP vectors whose mantissa buffer was (successfully) dynamically allocated have a flag set which indicates as much. This function can safely be called on any *bfp_complex_s16_t* which has not had its `flags` or `real` manually manipulated, including:

- *bfp_complex_s16_t* resulting from a successful call to *bfp_complex_s16_alloc()*
- *bfp_complex_s16_t* resulting from an unsuccessful call to *bfp_complex_s16_alloc()*

- *bfp_complex_s16_t* initialized with a call to *bfp_complex_s16_init()*

In the latter two cases, this function does nothing. In the former, the `real`, `imag`, `length` and `flags` fields of `vector` are cleared to zero.

See also:

bfp_complex_s16_alloc

Parameters

- **vector** − **[in]** BFP vector to be deallocated.

void **bfp_complex_s16_set**(*bfp_complex_s16_t* *a, const *complex_s16_t* b, const *exponent_t* exp)

Set all elements of a complex 16-bit BFP vector to a specified value.

The exponent of `a` is set to `exp`, and each element's mantissa is set to `b`.

After performing this operation, all elements will represent the same value $b \cdot 2^{exp}$.

`a` must have been initialized (see *bfp_complex_s16_init()*).

Parameters

- **a** − **[out]** BFP vector to update
- **b** − **[in]** New value each complex mantissa is set to
- **exp** − **[in]** New exponent for the BFP vector

void **bfp_complex_s16_use_exponent**(*bfp_complex_s16_t* *a, const *exponent_t* exp)

Modify a complex 16-bit BFP vector to use a specified exponent.

This function forces complex BFP vector $\bar{A}$ to use a specified exponent. The mantissa vector $\bar{a}$ will be bit-shifted left or right to compensate for the changed exponent.

This function can be used, for example, before calling a fixed-point arithmetic function to ensure the underlying mantissa vector has the needed Q-format. As another example, this may be useful when communicating with peripheral devices (e.g. via I2S) that require sample data to be in a specified format.

Note that this sets the *current* encoding, and does not *fix* the exponent permanently (i.e. subsequent operations may change the exponent as usual).

If the required fixed-point Q-format is QX.Y, where Y is the number of fractional bits in the resulting mantissas, then the associated exponent (and value for parameter `exp`) is -Y.

`a` points to input BFP vector $\bar{A}$, with complex mantissa vector $\bar{a}$ and exponent a_exp. `a` is updated in place to produce resulting BFP vector $\tilde{\bar{A}}$ with complex mantissa vector $\tilde{\bar{a}}$ and exponent $\tilde{a}\_exp$.

`exp` is $\tilde{a}\_exp$, the required exponent. $\Delta p = \tilde{a}\_exp - a\_exp$ is the required change in exponent.

If $\Delta p = 0$, the BFP vector is left unmodified.

If $\Delta p > 0$, the required exponent is larger than the current exponent and an arithmetic right-shift of Δp bits is applied to the mantissas $\bar{a}$. When applying a right-shift, precision may be lost by discarding the Δp least significant bits.

If $\Delta p < 0$, the required exponent is smaller than the current exponent and a left-shift of Δp bits is applied to the mantissas $\bar{a}$. When left-shifting, saturation logic will be applied such that any element that can't be represented exactly with the new exponent will saturate to the 16-bit saturation bounds.

The exponent and headroom of `a` are updated by this function.

Operation Performed

$$\Delta p = \tilde{a}\_exp - a\_exp$$
$$\tilde{a_k} \leftarrow sat_{16}(a_k \cdot 2^{-\Delta p})$$
$$\text{for } k \in 0 \ldots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{A} \text{ (in elements)}$$

Parameters

- **a** – **[inout]** Input BFP vector $\bar{A}$ / Output BFP vector $\bar{\tilde{A}}$

- **exp** – **[in]** The required exponent, $\tilde{a}\_exp$

headroom_t **bfp_complex_s16_headroom**(*bfp_complex_s16_t* *b)

Get the headroom of a complex 16-bit BFP vector.

The headroom of a complex vector is the number of bits that the real and imaginary parts of each of its elements can be left-shifted without losing any information. It conveys information about the range of values that vector may contain, which is useful for determining how best to preserve precision in potentially lossy block floating-point operations.

In a BFP context, headroom applies to mantissas only, not exponents.

In particular, if the complex 16-bit mantissa vector $\bar{x}$ has N bits of headroom, then for any element x_k of $\bar{x}$

$$-2^{15-N} \le Re\{x_k\} < 2^{15-N}$$

and

$$-2^{15-N} \le Im\{x_k\} < 2^{15-N}$$

And for any element $X_k = x_k \cdot 2^{x\_exp}$ of a complex BFP vector $\bar{X}$

$$-2^{15+x\_exp-N} \le Re\{X_k\} < 2^{15+x\_exp-N}$$

and

$$-2^{15+x\_exp-N} \le Im\{X_k\} < 2^{15+x\_exp-N}$$

This function determines the headroom of `b`, updates `b->hr` with that value, and then returns `b->hr`.

Parameters

- **b** – complex BFP vector to get the headroom of

Returns

Headroom of complex BFP vector `b`

void **bfp_complex_s16_shl**(*bfp_complex_s16_t* *a, const *bfp_complex_s16_t* *b, const *left_shift_t* b_shl)

Apply a left-shift to the mantissas of a complex 16-bit BFP vector.

Each complex mantissa of input BFP vector $\bar{B}$ is left-shifted `b_shl` bits and stored in the corresponding element of output BFP vector $\bar{A}$.

This operation can be used to add or remove headroom from a BFP vector.

`b_shr` is the number of bits that the real and imaginary parts of each mantissa will be left-shifted. This shift is signed and arithmetic, so negative values for `b_shl` will right-shift the mantissas.

`a` and `b` must have been initialized (see *bfp_complex_s16_init()*), and must be the same length.

This operation can be performed safely in-place on `b`.

Note that this operation bypasses the logic protecting the caller from saturation or underflows. Output values saturate to the symmetric 16-bit range (the open interval $(-2^{15}, 2^{15})$). To avoid saturation, `b_shl` should be no greater than the headroom of `b` (`b->hr`).

Operation Performed

$$Re\{a_k\} \leftarrow sat_{16}(\lfloor Re\{b_k\} \cdot 2^{b\_shl} \rfloor)$$
$$Im\{a_k\} \leftarrow sat_{16}(\lfloor Im\{b_k\} \cdot 2^{b\_shl} \rfloor)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$
$$\text{and } b_k \text{ and } a_k \text{ are the } k\text{th mantissas from } \bar{B} \text{ and } \bar{A} \text{ respectively}$$

Parameters

- **a** – **[out]** Complex output BFP vector $\bar{A}$
- **b** – **[in]** Complex input BFP vector $\bar{B}$
- **b_shl** – **[in]** Signed arithmetic left-shift to be applied to mantissas of $\bar{B}$.

void **bfp_complex_s16_real_mul**(*bfp_complex_s16_t* *a, const *bfp_complex_s16_t* *b, const *bfp_s16_t* *c)

Multiply a complex 16-bit BFP vector element-wise by a real 16-bit BFP vector.

Each complex output element A_k of complex output BFP vector $\bar{A}$ is set to the complex product of B_k and C_k, the corresponding elements of complex input BFP vector $\bar{B}$ and real input BFP vector $\bar{C}$ respectively.

a, b and c must have been initialized (see *bfp_complex_s16_init()* and *bfp_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$A_k \leftarrow B_k \cdot C_k$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **a** – **[out]** Output complex BFP vector $\bar{A}$
- **b** – **[in]** Input complex BFP vector $\bar{B}$
- **c** – **[in]** Input real BFP vector $\bar{C}$

void **bfp_complex_s16_mul**(*bfp_complex_s16_t* *a, const *bfp_complex_s16_t* *b, const *bfp_complex_s16_t* *c)

Multiply one complex 16-bit BFP vector element-wise another.

Each complex output element A_k of complex output BFP vector $\bar{A}$ is set to the complex product of B_k and C_k, the corresponding elements of complex input BFP vectors $\bar{B}$ and $\bar{C}$ respectively.

a, b and c must have been initialized (see *bfp_complex_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b or c.

Operation Performed

$$A_k \leftarrow B_k \cdot C_k$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **a** – **[out]** Output complex BFP vector $\bar{A}$
- **b** – **[in]** Input complex BFP vector $\bar{B}$
- **c** – **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s16_conj_mul**(*bfp_complex_s16_t* *a, const *bfp_complex_s16_t* *b, const *bfp_complex_s16_t* *c)

Multiply one complex 16-bit BFP vector element-wise by the complex conjugate of another.

Each complex output element A_k of complex output BFP vector $\bar{A}$ is set to the complex product of B_k, the corresponding element of complex input BFP vectors $\bar{B}$, and $(C_k)^*$, the complex conjugate of the corresponding element of complex input BFP vector $\bar{C}$.

Operation Performed

$$A_k \leftarrow B_k \cdot (C_k)^*$$
$$\text{for } k \in 0 \ldots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$
$$\text{and } (C_k)^* \text{ is the complex conjugate of } C_k$$

Parameters

- **a** – **[out]** Output complex BFP vector $\bar{A}$
- **b** – **[in]** Input complex BFP vector $\bar{B}$
- **c** – **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s16_macc**(*bfp_complex_s16_t* *acc, const *bfp_complex_s16_t* *b, const *bfp_complex_s16_t* *c)

Multiply one complex 16-bit BFP vector by another element-wise and add the result to a third vector.

Operation Performed

$$A_k \leftarrow A_k + (B_k \cdot C_k)$$
$$\text{for } k \in 0 \ldots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **acc** – **[inout]** Input/Output accumulator complex BFP vector $\bar{A}$
- **b** – **[in]** Input complex BFP vector $\bar{B}$
- **c** – **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s16_nmacc**(*bfp_complex_s16_t* *acc, const *bfp_complex_s16_t* *b, const *bfp_complex_s16_t* *c)

Multiply one complex 16-bit BFP vector by another element-wise and subtract the result from a third vector.

Operation Performed

$$A_k \leftarrow A_k - (B_k \cdot C_k)$$
$$\text{for } k \in 0 \ldots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **acc** − **[inout]** Input/Output accumulator complex BFP vector $\bar{A}$

- **b** − **[in]** Input complex BFP vector $\bar{B}$

- **c** − **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s16_conj_macc**(*bfp_complex_s16_t* *acc, const *bfp_complex_s16_t* *b, const *bfp_complex_s16_t* *c)

Multiply one complex 16-bit BFP vector by the complex conjugate of another element-wise and add the result to a third vector.

Operation Performed

$$A_k \leftarrow A_k + (B_k \cdot C_k^*)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$
$$\text{and } (C_k)^* \text{ is the complex conjugate of } C_k$$

Parameters

- **acc** − **[inout]** Input/Output accumulator complex BFP vector $\bar{A}$

- **b** − **[in]** Input complex BFP vector $\bar{B}$

- **c** − **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s16_conj_nmacc**(*bfp_complex_s16_t* *acc, const *bfp_complex_s16_t* *b, const *bfp_complex_s16_t* *c)

Multiply one complex 16-bit BFP vector by the complex conjugate of another element-wise and subtract the result from a third vector.

Operation Performed

$$A_k \leftarrow A_k - (B_k \cdot C_k^*)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$
$$\text{and } (C_k)^* \text{ is the complex conjugate of } C_k$$

Parameters

- **acc** − **[inout]** Input/Output accumulator complex BFP vector $\bar{A}$

- **b** − **[in]** Input complex BFP vector $\bar{B}$

- **c** − **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s16_real_scale**(*bfp_complex_s16_t* *a, const *bfp_complex_s16_t* *b, const float alpha)

Multiply a complex 16-bit BFP vector by a real scalar.

Each complex output element A_k of complex output BFP vector $\bar{A}$ is set to the complex product of B_k, the corresponding element of complex input BFP vector $\bar{B}$, and real scalar $\alpha \cdot 2^{\alpha\_exp}$, where α and α_exp are the mantissa and exponent respectively of parameter alpha. a and b must have been initialized (see *bfp_complex_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$\bar{A} \leftarrow \bar{B} \cdot (\alpha \cdot 2^{\alpha\_exp})$$

Parameters

- **a** − **[out]** Output complex BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$
- **alpha** − **[in]** Real scalar by which $\bar{B}$ is multiplied

void **bfp_complex_s16_scale**(*bfp_complex_s16_t* *a, const *bfp_complex_s16_t* *b, const *float_complex_s16_t* alpha)

Multiply a complex 16-bit BFP vector by a complex scalar.

Each complex output element A_k of complex output BFP vector $\bar{A}$ is set to the complex product of B_k, the corresponding element of complex input BFP vector $\bar{B}$, and complex scalar $\alpha \cdot 2^{\alpha\_exp}$, where α and α_exp are the complex mantissa and exponent respectively of parameter `alpha`.

`a` and `b` must have been initialized (see *bfp_complex_s16_init()*), and must be the same length.

This operation can be performed safely in-place on `b`.

Operation Performed

$$\bar{A} \leftarrow \bar{B} \cdot (\alpha \cdot 2^{\alpha\_exp})$$

Parameters

- **a** − **[out]** Output complex BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$
- **alpha** − **[in]** Complex scalar by which $\bar{B}$ is multiplied

void **bfp_complex_s16_add**(*bfp_complex_s16_t* *a, const *bfp_complex_s16_t* *b, const *bfp_complex_s16_t* *c)

Add one complex 16-bit BFP vector to another.

Each complex output element A_k of complex output BFP vector $\bar{A}$ is set to the sum of B_k and C_k, the corresponding elements of complex input BFP vectors $\bar{B}$ and $\bar{C}$ respectively.

`a`, `b` and `c` must have been initialized (see *bfp_complex_s16_init()*), and must be the same length.

This operation can be performed safely in-place on `b` or `c`.

Operation Performed

$$\bar{A} \leftarrow \bar{B} + \bar{C}$$

Parameters

- **a** − **[out]** Output complex BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$
- **c** − **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s16_add_scalar**(*bfp_complex_s16_t* *a, const *bfp_complex_s16_t* *b, const *float_complex_s16_t* c)

Add a complex scalar to a complex 16-bit BFP vector.

Add a real scalar c to input BFP vector $\bar{B}$ and store the result in BFP vector $\bar{A}$.

a, and b must have been initialized (see *bfp_complex_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$\bar{A} \leftarrow \bar{B} + c$$

Parameters

- **a** − **[out]** Output complex BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$
- **c** − **[in]** Input complex scalar c

void **bfp_complex_s16_sub**(*bfp_complex_s16_t* *a, const *bfp_complex_s16_t* *b, const *bfp_complex_s16_t* *c)

Subtract one complex 16-bit BFP vector from another.

Each complex output element A_k of complex output BFP vector $\bar{A}$ is set to the difference between B_k and C_k, the corresponding elements of complex input BFP vectors $\bar{B}$ and $\bar{C}$ respectively.

a, b and c must have been initialized (see *bfp_complex_s16_init()*), and must be the same length.

This operation can be performed safely in-place on b or c.

Operation Performed

$$\bar{A} \leftarrow \bar{B} - \bar{C}$$

Parameters

- **a** − **[out]** Output complex BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$
- **c** − **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s16_to_bfp_complex_s32**(*bfp_complex_s32_t* *a, const *bfp_complex_s16_t* *b)

Convert a complex 16-bit BFP vector to a complex 32-bit BFP vector.

Each complex 32-bit output element A_k of complex output BFP vector $\bar{A}$ is set to the value of B_k, the corresponding element of complex 16-bit input BFP vector $\bar{B}$, sign-extended to 32 bits.

a and b must have been initialized (see *bfp_complex_s32_init()* and *bfp_complex_s16_init()*), and must be the same length.

Operation Performed

$$A_k \overset{32-bit}{\longleftarrow} B_k$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **a** − **[out]** Output complex 32-bit BFP vector $\bar{A}$
- **b** − **[in]** Input complex 16-bit BFP vector $\bar{B}$

void **bfp_complex_s16_squared_mag**(*bfp_s16_t* *a, const *bfp_complex_s16_t* *b)

Get the squared magnitude of each element of a complex 16-bit BFP vector.

Each element A_k of real output BFP vector $\bar{A}$ is set to the squared magnitude of B_k, the corresponding element of complex input BFP vector $\bar{B}$.

a and b must have been initialized (see *bfp_s16_init() bfp_complex_s16_init()*), and must be the same length.

Operation Performed

$$A_k \leftarrow B_k \cdot (B_k)^*$$
$$\text{for } k \in 0 \ldots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$
$$\text{and } (B_k)^* \text{ is the complex conjugate of } B_k$$

Parameters

- **a** − **[out]** Output real BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$

void **bfp_complex_s16_mag**(*bfp_s16_t* *a, const *bfp_complex_s16_t* *b)

Get the magnitude of each element of a complex 16-bit BFP vector.

Each element A_k of real output BFP vector $\bar{A}$ is set to the magnitude of B_k, the corresponding element of complex input BFP vector $\bar{B}$.

a and b must have been initialized (see *bfp_s16_init() bfp_complex_s16_init()*), and must be the same length.

Operation Performed

$$A_k \leftarrow |B_k|$$
$$\text{for } k \in 0 \ldots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **a** − **[out]** Output real BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$

float_complex_s32_t **bfp_complex_s16_sum**(const *bfp_complex_s16_t* *b)

Get the sum of elements of a complex 16-bit BFP vector.

The elements of complex input BFP vector $\bar{B}$ are summed together. The result is a complex 32-bit floating-point scalar a, which is returned.

b must have been initialized (see *bfp_complex_s16_init()*).

Operation Performed

$$a \leftarrow \sum_{k=0}^{N-1} \left(b_k \cdot 2^{B\_exp} \right)$$

for $k \in 0 \dots (N-1)$

where N is the length of $\bar{B}$

Parameters

- **b** − **[in]** Input complex BFP vector $\bar{B}$

Returns

a, the sum of vector $\bar{B}$'s elements

void **bfp_complex_s16_conjugate**(*bfp_complex_s16_t* *a, const *bfp_complex_s16_t* *b)

Get the complex conjugate of each element of a complex 16-bit BFP vector.

Each element A_k of complex output BFP vector $\bar{A}$ is set to the complex conjugate of B_k, the corresponding element of complex input BFP vector $\bar{B}$.

Operation Performed

$$A_k \leftarrow B_k^*$$

for $k \in 0 \dots (N-1)$

where N is the length of $\bar{B}$ and $\bar{C}$

and B_k^* is the complex conjugate of B_k

Parameters

- **a** − **[out]** Output complex BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$

float_s64_t **bfp_complex_s16_energy**(const *bfp_complex_s16_t* *b)

Get the energy of a complex 16-bit BFP vector.

The energy of a complex 16-bit BFP vector here is the sum of the squared magnitudes of each of the vector's elements.

Operation Performed

$$a \leftarrow \sum_{k=0}^{N-1} \left(\left| b_k \cdot 2^{B\_exp} \right|^2 \right)$$

for $k \in 0 \dots (N-1)$

where N is the length of $\bar{B}$

Parameters

- **b** − **[in]** Input complex BFP vector $\bar{B}$

Returns

a, the energy of vector $\bar{B}$

4.2.5 Complex 32-Bit Block Floating-Point API

group `bfp_complex_s32_api`

Functions

void **bfp_complex_s32_init**(*bfp_complex_s32_t* *a, *complex_s32_t* *data, const *exponent_t* exp, const unsigned length, const unsigned calc_hr)

Initialize a 32-bit complex BFP vector.

This function initializes each of the fields of `a`.

Unlike `bfp_complex_s16_t`, complex 32-bit BFP vectors use a single buffer to store the real and imaginary parts of each mantissa, such that the imaginary part of element `k` follows the real part of element `k` in memory. `data` points to the memory buffer used to store elements of the vector, and must be at least `length * 8` bytes long.

`exp` is the exponent assigned to the BFP vector. The logical value associated with the `k`th complex element of the vector after initialization will be $(data_{2k} + i \cdot data_{2k+1}) \cdot 2^{exp}$.

If `calc_hr` is false, `a->hr` is initialized to 0. Otherwise, the headroom of the the BFP vector is calculated and used to initialize `a->hr`.

Parameters

- **a** – **[out]** BFP vector struct to initialize
- **data** – **[in]** `complex_s32_t` buffer used to back `a`
- **exp** – **[in]** Exponent of BFP vector
- **length** – **[in]** Number of elements in BFP vector
- **calc_hr** – **[in]** Boolean indicating whether the HR of the BFP vector should be calculated

bfp_complex_s32_t **bfp_complex_s32_alloc**(const unsigned length)

Dynamically allocate a complex 32-bit BFP vector from the heap.

If allocation was unsuccessful, the `data` field of the returned vector will be NULL, and the `length` field will be zero. Otherwise, `data` will point to the allocated memory and the `length` field will be the user-specified length. The `length` argument must not be zero.

Neither the BFP exponent, headroom, nor the elements of the allocated mantissa vector are set by this function. To set the BFP vector elements to a known value, use *bfp_complex_s32_set()* on the retuned BFP vector.

BFP vectors allocated using this function must be deallocated using *bfp_complex_s32_dealloc()* to avoid a memory leak.

To initialize a BFP vector using static memory allocation, use *bfp_complex_s32_init()* instead.

See also:

bfp_complex_s32_dealloc, *bfp_complex_s32_init*

Note: Dynamic allocation of BFP vectors relies on allocation from the heap, and offers no guarantees about the execution time. Use of this function in any time-critical section of code is highly discouraged.

Parameters

- **length** – **[in]** The length of the BFP vector to be allocated (in elements)

Returns

Complex 32-bit BFP vector

void **bfp_complex_s32_dealloc**(*bfp_complex_s32_t* *vector)

Deallocate a complex 32-bit BFP vector allocated by *bfp_complex_s32_alloc()*.

Use this function to free the heap memory allocated by *bfp_complex_s32_alloc()*.

BFP vectors whose mantissa buffer was (successfully) dynamically allocated have a flag set which indicates as much. This function can safely be called on any *bfp_complex_s32_t* which has not had its `flags` or `data` manually manipulated, including:

- *bfp_complex_s32_t* resulting from a successful call to *bfp_complex_s32_alloc()*
- *bfp_complex_s32_t* resulting from an unsuccessful call to *bfp_complex_s32_alloc()*
- *bfp_complex_s32_t* initialized with a call to *bfp_complex_s32_init()*

In the latter two cases, this function does nothing. In the former, the `data`, `length` and `flags` fields of `vector` are cleared to zero.

See also:

bfp_complex_s32_alloc

Parameters

- **vector** – **[in]** BFP vector to be deallocated.

void **bfp_complex_s32_set**(*bfp_complex_s32_t* *a, const *complex_s32_t* b, const *exponent_t* exp)

Set all elements of a complex 32-bit BFP vector to a specified value.

The exponent of `a` is set to `exp`, and each element's mantissa is set to `b`.

After performing this operation, all elements will represent the same value $b \cdot 2^{exp}$.

`a` must have been initialized (see *bfp_complex_s32_init()*).

Parameters

- **a** – **[out]** BFP vector to update
- **b** – **[in]** New value each complex mantissa is set to
- **exp** – **[in]** New exponent for the BFP vector

void **bfp_complex_s32_use_exponent**(*bfp_complex_s32_t* *a, const *exponent_t* exp)

Modify a complex 32-bit BFP vector to use a specified exponent.

This function forces complex BFP vector $\bar{A}$ to use a specified exponent. The mantissa vector $\bar{a}$ will be bit-shifted left or right to compensate for the changed exponent.

This function can be used, for example, before calling a fixed-point arithmetic function to ensure the underlying mantissa vector has the needed Q-format. As another example, this may be useful when communicating with peripheral devices (e.g. via I2S) that require sample data to be in a specified format.

Note that this sets the *current* encoding, and does not *fix* the exponent permanently (i.e. subsequent operations may change the exponent as usual).

If the required fixed-point Q-format is `QX.Y`, where `Y` is the number of fractional bits in the resulting mantissas, then the associated exponent (and value for parameter `exp`) is `-Y`.

`a` points to input BFP vector $\bar{A}$, with complex mantissa vector $\bar{a}$ and exponent a_exp. `a` is updated in place to produce resulting BFP vector $\tilde{A}$ with complex mantissa vector $\tilde{a}$ and exponent $\tilde{a}\_exp$.

exp is $\tilde{a}\_exp$, the required exponent. $\Delta p = \tilde{a}\_exp - a\_exp$ is the required change in exponent.

If $\Delta p = 0$, the BFP vector is left unmodified.

If $\Delta p > 0$, the required exponent is larger than the current exponent and an arithmetic right-shift of Δp bits is applied to the mantissas $\bar{a}$. When applying a right-shift, precision may be lost by discarding the Δp least significant bits.

If $\Delta p < 0$, the required exponent is smaller than the current exponent and a left-shift of Δp bits is applied to the mantissas $\bar{a}$. When left-shifting, saturation logic will be applied such that any element that can't be represented exactly with the new exponent will saturate to the 32-bit saturation bounds.

The exponent and headroom of a are updated by this function.

Operation Performed

$$\Delta p = \tilde{a}\_exp - a\_exp$$
$$\tilde{a_k} \leftarrow sat_{32}(a_k \cdot 2^{-\Delta p})$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{A} \text{ (in elements)}$$

Parameters

- **a** – **[inout]** Input BFP vector $\bar{A}$ / Output BFP vector $\tilde{A}$

- **exp** – **[in]** The required exponent, $\tilde{a}\_exp$

headroom_t **bfp_complex_s32_headroom**(*bfp_complex_s32_t* *b)

Get the headroom of a complex 32-bit BFP vector.

The headroom of a complex vector is the number of bits that the real and imaginary parts of each of its elements can be left-shifted without losing any information. It conveys information about the range of values that vector may contain, which is useful for determining how best to preserve precision in potentially lossy block floating-point operations.

In a BFP context, headroom applies to mantissas only, not exponents.

In particular, if the complex 32-bit mantissa vector $\bar{x}$ has N bits of headroom, then for any element x_k of $\bar{x}$

$-2^{31-N} \le Re\{x_k\} < 2^{31-N}$

and

$-2^{31-N} \le Im\{x_k\} < 2^{31-N}$

And for any element $X_k = x_k \cdot 2^{x\_exp}$ of a complex BFP vector $\bar{X}$

$-2^{31+x\_exp-N} \le Re\{X_k\} < 2^{31+x\_exp-N}$

and

$-2^{31+x\_exp-N} \le Im\{X_k\} < 2^{31+x\_exp-N}$

This function determines the headroom of b, updates b->hr with that value, and then returns b->hr.

Parameters

- **b** – complex BFP vector to get the headroom of

Returns

Headroom of complex BFP vector b

void **bfp_complex_s32_shl**(*bfp_complex_s32_t* *a, const *bfp_complex_s32_t* *b, const *left_shift_t* b_shl)

Apply a left-shift to the mantissas of a complex 32-bit BFP vector.

Each complex mantissa of input BFP vector $\bar{B}$ is left-shifted b_shl bits and stored in the corresponding element of output BFP vector $\bar{A}$.

This operation can be used to add or remove headroom from a BFP vector.

b_shl is the number of bits that the real and imaginary parts of each mantissa will be left-shifted. This shift is signed and arithmetic, so negative values for b_shl will right-shift the mantissas.

a and b must have been initialized (see *bfp_complex_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Note that this operation bypasses the logic protecting the caller from saturation or underflows. Output values saturate to the symmetric 32-bit range (the open interval $(-2^{31}, 2^{31})$). To avoid saturation, b_shl should be no greater than the headroom of b (b->hr).

Operation Performed

$$Re\{a_k\} \leftarrow sat_{32}(\lfloor Re\{b_k\} \cdot 2^{b\_shl}\rfloor)$$
$$Im\{a_k\} \leftarrow sat_{32}(\lfloor Im\{b_k\} \cdot 2^{b\_shl}\rfloor)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$
$$\text{and } b_k \text{ and } a_k \text{ are the } k\text{th mantissas from } \bar{B} \text{ and } \bar{A} \text{ respectively}$$

Parameters

- **a** – **[out]** Complex output BFP vector $\bar{A}$
- **b** – **[in]** Complex input BFP vector $\bar{B}$
- **b_shl** – **[in]** Signed arithmetic left-shift to be applied to mantissas of $\bar{B}$.

void **bfp_complex_s32_real_mul**(*bfp_complex_s32_t* *a, const *bfp_complex_s32_t* *b, const *bfp_s32_t* *c)

Multiply a complex 32-bit BFP vector element-wise by a real 32-bit BFP vector.

Each complex output element A_k of complex output BFP vector $\bar{A}$ is set to the complex product of B_k and C_k, the corresponding elements of complex input BFP vector $\bar{B}$ and real input BFP vector $\bar{C}$ respectively.

a, b and c must have been initialized (see *bfp_complex_s32_init()* and *bfp_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$A_k \leftarrow B_k \cdot C_k$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **a** – **[out]** Output complex BFP vector $\bar{A}$
- **b** – **[in]** Input complex BFP vector $\bar{B}$

- **c** − **[in]** Input real BFP vector $\bar{C}$

void **bfp_complex_s32_mul**(*bfp_complex_s32_t* *a, const *bfp_complex_s32_t* *b, const *bfp_complex_s32_t* *c)

Multiply one complex 32-bit BFP vector element-wise by another.

Each complex output element A_k of complex output BFP vector $\bar{A}$ is set to the complex product of B_k and C_k, the corresponding elements of complex input BFP vectors $\bar{B}$ and $\bar{C}$ respectively.

a, b and c must have been initialized (see *bfp_complex_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b or c.

Operation Performed

$$A_k \leftarrow B_k \cdot C_k$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **a** − **[out]** Output complex BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$
- **c** − **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s32_conj_mul**(*bfp_complex_s32_t* *a, const *bfp_complex_s32_t* *b, const *bfp_complex_s32_t* *c)

Multiply one complex 32-bit BFP vector element-wise by the complex conjugate of another.

Each complex output element A_k of complex output BFP vector $\bar{A}$ is set to the complex product of B_k, the corresponding element of complex input BFP vectors $\bar{B}$, and $(C_k)^*$, the complex conjugate of the corresponding element of complex input BFP vector $\bar{C}$.

a, b and c must have been initialized (see *bfp_complex_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b or c.

Operation Performed

$$A_k \leftarrow B_k \cdot (C_k)^*$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$
$$\text{and } (C_k)^* \text{ is the complex conjugate of } C_k$$

Parameters

- **a** − **[out]** Output complex BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$
- **c** − **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s32_macc**(*bfp_complex_s32_t* *acc, const *bfp_complex_s32_t* *b, const *bfp_complex_s32_t* *c)

Multiply one complex 32-bit BFP vector by another element-wise and add the result to a third vector.

Operation Performed

$$A_k \leftarrow A_k + (B_k \cdot C_k)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **acc** − **[inout]** Input/Output accumulator complex BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$
- **c** − **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s32_nmacc**(*bfp_complex_s32_t* *acc, const *bfp_complex_s32_t* *b, const *bfp_complex_s32_t* *c)

Multiply one complex 32-bit BFP vector by another element-wise and subtract the result from a third vector.

Operation Performed

$$A_k \leftarrow A_k - (B_k \cdot C_k)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$

Parameters

- **acc** − **[inout]** Input/Output accumulator complex BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$
- **c** − **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s32_conj_macc**(*bfp_complex_s32_t* *acc, const *bfp_complex_s32_t* *b, const *bfp_complex_s32_t* *c)

Multiply one complex 32-bit BFP vector by the complex conjugate of another element-wise and add the result to a third vector.

Operation Performed

$$A_k \leftarrow A_k + (B_k \cdot C_k^*)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$
$$\text{and } (C_k)^* \text{ is the complex conjugate of } C_k$$

Parameters

- **acc** − **[inout]** Input/Output accumulator complex BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$
- **c** − **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s32_conj_nmacc**(*bfp_complex_s32_t* *acc, const *bfp_complex_s32_t* *b, const *bfp_complex_s32_t* *c)

Multiply one complex 32-bit BFP vector by the complex conjugate of another element-wise and subtract the result from a third vector.

Operation Performed

$$A_k \leftarrow A_k - (B_k \cdot C_k^*)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$
$$\text{and } (C_k)^* \text{ is the complex conjugate of } C_k$$

Parameters

- **acc** − **[inout]** Input/Output accumulator complex BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$
- **c** − **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s32_real_scale**(*bfp_complex_s32_t* *a, const *bfp_complex_s32_t* *b, const *float_s32_t* alpha)

Multiply a complex 32-bit BFP vector by a real scalar.

Each complex output element A_k of complex output BFP vector $\bar{A}$ is set to the complex product of B_k, the corresponding element of complex input BFP vector $\bar{B}$, and real scalar $\alpha \cdot 2^{\alpha\_exp}$, where α and α_exp are the mantissa and exponent respectively of parameter alpha.

a and b must have been initialized (see *bfp_complex_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$\bar{A} \leftarrow \bar{B} \cdot (\alpha \cdot 2^{\alpha\_exp})$$

Parameters

- **a** − **[out]** Output complex BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$
- **alpha** − **[in]** Real scalar by which $\bar{B}$ is multiplied

void **bfp_complex_s32_scale**(*bfp_complex_s32_t* *a, const *bfp_complex_s32_t* *b, const *float_complex_s32_t* alpha)

Multiply a complex 32-bit BFP vector by a complex scalar.

Each complex output element A_k of complex output BFP vector $\bar{A}$ is set to the complex product of B_k, the corresponding element of complex input BFP vector $\bar{B}$, and complex scalar $\alpha \cdot 2^{\alpha\_exp}$, where α and α_exp are the complex mantissa and exponent respectively of parameter alpha.

a and b must have been initialized (see *bfp_complex_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$\bar{A} \leftarrow \bar{B} \cdot (\alpha \cdot 2^{\alpha\_exp})$$

Parameters

- **a** − **[out]** Output complex BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$
- **alpha** − **[in]** Complex scalar by which $\bar{B}$ is multiplied

void **bfp_complex_s32_add**(*bfp_complex_s32_t* *a, const *bfp_complex_s32_t* *b, const *bfp_complex_s32_t* *c)

Add one complex 32-bit BFP vector to another.

Each complex output element A_k of complex output BFP vector $\bar{A}$ is set to the sum of B_k and C_k, the corresponding elements of complex input BFP vectors $\bar{B}$ and $\bar{C}$ respectively.

a, b and c must have been initialized (see *bfp_complex_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b or c.

Operation Performed

$$\bar{A} \leftarrow \bar{B} + \bar{C}$$

Parameters

- **a** – **[out]** Output complex BFP vector $\bar{A}$
- **b** – **[in]** Input complex BFP vector $\bar{B}$
- **c** – **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s32_add_scalar**(*bfp_complex_s32_t* *a, const *bfp_complex_s32_t* *b, const *float_complex_s32_t* c)

Add a complex scalar to a complex 32-bit BFP vector.

Add a real scalar c to input BFP vector $\bar{B}$ and store the result in BFP vector $\bar{A}$.

a, and b must have been initialized (see *bfp_complex_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b.

Operation Performed

$$\bar{A} \leftarrow \bar{B} + c$$

Parameters

- **a** – **[out]** Output complex BFP vector $\bar{A}$
- **b** – **[in]** Input complex BFP vector $\bar{B}$
- **c** – **[in]** Input complex scalar c

void **bfp_complex_s32_sub**(*bfp_complex_s32_t* *a, const *bfp_complex_s32_t* *b, const *bfp_complex_s32_t* *c)

Subtract one complex 32-bit BFP vector from another.

Each complex output element A_k of complex output BFP vector $\bar{A}$ is set to the difference between B_k and C_k, the corresponding elements of complex input BFP vectors $\bar{B}$ and $\bar{C}$ respectively.

a, b and c must have been initialized (see *bfp_complex_s32_init()*), and must be the same length.

This operation can be performed safely in-place on b or c.

Operation Performed

$$\bar{A} \leftarrow \bar{B} - \bar{C}$$

Parameters

- **a** − **[out]** Output complex BFP vector $\bar{A}$

- **b** − **[in]** Input complex BFP vector $\bar{B}$

- **c** − **[in]** Input complex BFP vector $\bar{C}$

void **bfp_complex_s32_to_bfp_complex_s16**(*bfp_complex_s16_t* *a, const *bfp_complex_s32_t* *b)

Convert a complex 32-bit BFP vector to a complex 16-bit BFP vector.

Each complex 16-bit output element A_k of complex output BFP vector $\bar{A}$ is set to the value of B_k, the corresponding element of complex 32-bit input BFP vector $\bar{B}$, with its bit-depth reduced to 16 bits.

a and b must have been initialized (see *bfp_complex_s16_init()* and *bfp_complex_s32_init()*), and must be the same length.

This function preserves as much precision as possible.

Operation Performed

$$A_k \overset{16-bit}{\longleftarrow} B_k$$
$$\text{for } k \in 0 \ldots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **a** − **[out]** Output complex 16-bit BFP vector $\bar{A}$

- **b** − **[in]** Input complex 32-bit BFP vector $\bar{B}$

void **bfp_complex_s32_squared_mag**(*bfp_s32_t* *a, const *bfp_complex_s32_t* *b)

Get the squared magnitude of each element of a complex 32-bit BFP vector.

Each element A_k of real output BFP vector $\bar{A}$ is set to the squared magnitude of B_k, the corresponding element of complex input BFP vector $\bar{B}$.

a and b must have been initialized (see *bfp_s32_init() bfp_complex_s32_init()*), and must be the same length.

Operation Performed

$$A_k \leftarrow B_k \cdot (B_k)^*$$
$$\text{for } k \in 0 \ldots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$
$$\text{and } (B_k)^* \text{ is the complex conjugate of } B_k$$

Parameters

- **a** − **[out]** Output real BFP vector $\bar{A}$

- **b** − **[in]** Input complex BFP vector $\bar{B}$

void **bfp_complex_s32_mag**(*bfp_s32_t* *a, const *bfp_complex_s32_t* *b)

Get the magnitude of each element of a complex 32-bit BFP vector.

Each element A_k of real output BFP vector $\bar{A}$ is set to the magnitude of B_k, the corresponding element of complex input BFP vector $\bar{B}$.

a and b must have been initialized (see *bfp_s32_init() bfp_complex_s32_init()*), and must be the same length.

Operation Performed

$$A_k \leftarrow |B_k|$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **a** − **[out]** Output real BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$

float_complex_s64_t **bfp_complex_s32_sum**(const *bfp_complex_s32_t* *b)

Get the sum of elements of a complex 32-bit BFP vector.

The elements of complex input BFP vector $\bar{B}$ are summed together. The result is a complex 64-bit floating-point scalar a, which is returned.

b must have been initialized (see *bfp_complex_s32_init()*).

Operation Performed

$$a \leftarrow \sum_{k=0}^{N-1} \left(b_k \cdot 2^{B\_exp} \right)$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **b** − **[in]** Input complex BFP vector $\bar{B}$

Returns

a, the sum of vector $\bar{B}$'s elements

void **bfp_complex_s32_conjugate**(*bfp_complex_s32_t* *a, const *bfp_complex_s32_t* *b)

Get the complex conjugate of each element of a complex 32-bit BFP vector.

Each element A_k of complex output BFP vector $\bar{A}$ is set to the complex conjugate of B_k, the corresponding element of complex input BFP vector $\bar{B}$.

Operation Performed

$$A_k \leftarrow B_k^*$$
$$\text{for } k \in 0 \dots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B} \text{ and } \bar{C}$$
$$\text{and } B_k^* \text{ is the complex conjugate of } B_k$$

Parameters

- **a** − **[out]** Output complex BFP vector $\bar{A}$
- **b** − **[in]** Input complex BFP vector $\bar{B}$

float_s64_t **bfp_complex_s32_energy**(const *bfp_complex_s32_t* *b)

Get the energy of a complex 32-bit BFP vector.

The energy of a complex 32-bit BFP vector here is the sum of the squared magnitudes of each of the vector's elements.

$$a \leftarrow \sum_{k=0}^{N-1} \left(\left| b_k \cdot 2^{B\_exp} \right|^2 \right)$$
$$\text{for } k \in 0 \ldots (N-1)$$
$$\text{where } N \text{ is the length of } \bar{B}$$

Parameters

- **b** − **[in]** Input complex BFP vector $\bar{B}$

Returns

a, the energy of vector $\bar{B}$

void **bfp_complex_s32_make**(*bfp_complex_s32_t* *a, const *bfp_s32_t* *b, const *bfp_s32_t* *c)

Create complex 32-bit BFP vector from real and imaginary parts.

Create a complex 32-bit BFP vector as the sum of a real vector $\bar{B}$ and imaginary vector $\bar{C}i$.

a, b and c must have been initialized (see *bfp_complex_s32_init()* and *bfp_s32_init()*), must be the same length. &a->data[0] must be a double-word-aligned address.

Operation Performed

$$\bar{A} \leftarrow \bar{B} + \bar{C}i$$

Parameters

- **a** − **[out]** Complex BFP output vector $\bar{A}$
- **b** − **[in]** Real BFP input vector $\bar{B}$
- **c** − **[in]** Real BFP input vector $\bar{C}$

void **bfp_complex_s32_real_part**(*bfp_s32_t* *a, const *bfp_complex_s32_t* *b)

Extract the real part of a complex 32-bit BFP vector.

This function populates the real 32-bit BFP vector $\bar{A}$ with the real part of complex 32-bit BFP vector $\bar{B}$.

&b->data[0] must be a double-word-aligned address.

Operation Performed

$$\bar{A} \leftarrow Real\{\bar{B}\}$$

Parameters

- **a** − **[out]** Real BFP output vector $\bar{A}$
- **b** − **[in]** Complex BFP input vector $\bar{B}$

void **bfp_complex_s32_imag_part**(*bfp_s32_t* *a, const *bfp_complex_s32_t* *b)

Extract the imaginary part of a complex 32-bit BFP vector.

This function populates the real 32-bit BFP vector $\bar{A}$ with the imaginary part of complex 32-bit BFP vector $\bar{B}$.

&b->data[0] must be a double-word-aligned address.

Operation Performed

$$\bar{A} \leftarrow Imag\{\bar{B}\}$$

Parameters

- **a** − **[out]** Real BFP output vector $\bar{A}$
- **b** − **[in]** Complex BFP input vector $\bar{B}$

4.3 Discrete Cosine Transform API

4.3.1 DCT API Quick Reference

Note: The forward DCTs are Type-II. The inverse of the Type-II DCT is the Type-III DCT, so Type-II and Type-III are supported here.

Table 4.8: DCT Functions - Quick Reference

Brief	Forward Function	Inverse Function
6-point DCT	dct6_forward()	dct6_inverse()
8-point DCT	dct8_forward()	dct8_inverse()
12-point DCT	dct12_forward()	dct12_inverse()
16-point DCT	dct16_forward()	dct16_inverse()
24-point DCT	dct24_forward()	dct24_inverse()
32-point DCT	dct32_forward()	dct32_inverse()
48-point DCT	dct48_forward()	dct48_inverse()
64-point DCT	dct64_forward()	dct64_inverse()
8-by-8 2-dimensional DCT	dct8x8_forward()	dct8x8_inverse()

Functions

void **dct6_forward**(int32_t y[6], const int32_t x[6])

6-point 32-bit forward DCT.

This function performs a 6-point forward type-II DCT on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result. To avoid possible overflow or saturation, output $\bar{y}$ is scaled down by a factor of 2^4 (see *dct6_exp*).

This operation may be safely performed in-place if x and y point to the same vector.

x and y must point to 8-byte-aligned addresses.

Operation Performed

$$y_k \leftarrow \frac{1}{2^4} \left(2 \sum_{n=0}^{N-1} x_n \cos\left(k\pi \frac{2n+1}{2N} \right) \right)$$
$$\text{for } k = 0, 1, \ldots, (N-1)$$
$$\text{with } N = 6$$

Parameters

- **y** − **[out]** Output vector $\bar{y}$
- **x** − **[in]** Input vector $\bar{x}$

Throws ET_LOAD_STORE
Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

void **dct8_forward**(int32_t y[8], const int32_t x[8])

8-point 32-bit forward DCT.

This function performs a 8-point forward type-II DCT on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result. To avoid possible overflow or saturation, output $\bar{y}$ is scaled down by a factor of 2^4 (see *dct8_exp*).

This operation may be safely performed in-place if x and y point to the same vector.

x and y must point to 8-byte-aligned addresses.

Operation Performed

$$y_k \leftarrow \frac{1}{2^4} \left(2 \sum_{n=0}^{N-1} x_n \cos\left(k\pi \frac{2n+1}{2N} \right) \right)$$
$$\text{for } k = 0, 1, \ldots, (N-1)$$
$$\text{with } N = 8$$

Parameters

- **y** − **[out]** Output vector $\bar{y}$
- **x** − **[in]** Input vector $\bar{x}$

Throws ET_LOAD_STORE
Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

void **dct12_forward**(int32_t y[12], const int32_t x[12])

  12-point 32-bit forward DCT.

  This function performs a 12-point forward type-II DCT on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result. To avoid possible overflow or saturation, output $\bar{y}$ is scaled down by a factor of 2^7 (see *dct12_exp*).

  This operation may be safely performed in-place if x and y point to the same vector.

  x and y must point to 8-byte-aligned addresses.

  Operation Performed

  $$y_k \leftarrow \frac{1}{2^7} \left(2 \sum_{n=0}^{N-1} x_n \cos\left(k\pi \frac{2n+1}{2N} \right) \right)$$
  $$\text{for } k = 0, 1, \ldots, (N-1)$$
  $$\text{with } N = 12$$

  Parameters

  - **y** − **[out]** Output vector $\bar{y}$
  - **x** − **[in]** Input vector $\bar{x}$

  Throws ET_LOAD_STORE
    Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

void **dct16_forward**(int32_t y[16], const int32_t x[16])

  16-point 32-bit forward DCT.

  This function performs a 16-point forward type-II DCT on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result. To avoid possible overflow or saturation, output $\bar{y}$ is scaled down by a factor of 2^7 (see *dct16_exp*).

  This operation may be safely performed in-place if x and y point to the same vector.

  x and y must point to 8-byte-aligned addresses.

  Operation Performed

  $$y_k \leftarrow \frac{1}{2^7} \left(2 \sum_{n=0}^{N-1} x_n \cos\left(k\pi \frac{2n+1}{2N} \right) \right)$$
  $$\text{for } k = 0, 1, \ldots, (N-1)$$
  $$\text{with } N = 16$$

  Parameters

  - **y** − **[out]** Output vector $\bar{y}$
  - **x** − **[in]** Input vector $\bar{x}$

  Throws ET_LOAD_STORE
    Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

void **dct24_forward**(int32_t y[24], const int32_t x[24])

  24-point 32-bit forward DCT.

  This function performs a 24-point forward type-II DCT on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result. To avoid possible overflow or saturation, output $\bar{y}$ is scaled down by a factor of 2^{10} (see *dct24_exp*).

This operation may be safely performed in-place if x and y point to the same vector.

x and y must point to 8-byte-aligned addresses.

Operation Performed

$$y_k \leftarrow \frac{1}{2^{10}} \left(2 \sum_{n=0}^{N-1} x_n \cos \left(k\pi \frac{2n+1}{2N} \right) \right)$$

$$\text{for } k = 0, 1, \dots, (N-1)$$

$$\text{with } N = 24$$

Parameters

- **y** − **[out]** Output vector $\bar{y}$
- **x** − **[in]** Input vector $\bar{x}$

Throws ET_LOAD_STORE
    Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

void **dct32_forward**(int32_t y[32], const int32_t x[32])

32-point 32-bit forward DCT.

This function performs a 32-point forward type-II DCT on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result. To avoid possible overflow or saturation, output $\bar{y}$ is scaled down by a factor of 2^{10} (see *dct32_exp*).

This operation may be safely performed in-place if x and y point to the same vector.

x and y must point to 8-byte-aligned addresses.

Operation Performed

$$y_k \leftarrow \frac{1}{2^{10}} \left(2 \sum_{n=0}^{N-1} x_n \cos \left(k\pi \frac{2n+1}{2N} \right) \right)$$

$$\text{for } k = 0, 1, \dots, (N-1)$$

$$\text{with } N = 32$$

Parameters

- **y** − **[out]** Output vector $\bar{y}$
- **x** − **[in]** Input vector $\bar{x}$

Throws ET_LOAD_STORE
    Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

void **dct48_forward**(int32_t y[48], const int32_t x[48])

48-point 32-bit forward DCT.

This function performs a 48-point forward type-II DCT on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result. To avoid possible overflow or saturation, output $\bar{y}$ is scaled down by a factor of 2^{13} (see *dct48_exp*).

This operation may be safely performed in-place if x and y point to the same vector.

x and y must point to 8-byte-aligned addresses.

Operation Performed

$$y_k \leftarrow \frac{1}{2^{13}} \left(2 \sum_{n=0}^{N-1} x_n \cos\left(k\pi \frac{2n+1}{2N} \right) \right)$$

for $k = 0, 1, \ldots, (N-1)$

with $N = 48$

Parameters

- **y** − **[out]** Output vector $\bar{y}$
- **x** − **[in]** Input vector $\bar{x}$

Throws ET_LOAD_STORE

Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

void **dct64_forward**(int32_t y[64], const int32_t x[64])

64-point 32-bit forward DCT.

This function performs a 64-point forward type-II DCT on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result. To avoid possible overflow or saturation, output $\bar{y}$ is scaled down by a factor of 2^{13} (see *dct64_exp*).

This operation may be safely performed in-place if x and y point to the same vector.

x and y must point to 8-byte-aligned addresses.

Operation Performed

$$y_k \leftarrow \frac{1}{2^{13}} \left(2 \sum_{n=0}^{N-1} x_n \cos\left(k\pi \frac{2n+1}{2N} \right) \right)$$

for $k = 0, 1, \ldots, (N-1)$

with $N = 64$

Parameters

- **y** − **[out]** Output vector $\bar{y}$
- **x** − **[in]** Input vector $\bar{x}$

Throws ET_LOAD_STORE

Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

void **dct6_inverse**(int32_t y[6], const int32_t x[6])

6-point 32-bit inverse DCT.

This function performs a 6-point inverse DCT (same as type-III DCT) on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result.

This operation may be safely performed in-place if x and y point to the same vector.

x and y must point to 8-byte-aligned addresses.

Operation Performed

$$y_k \leftarrow \frac{1}{N} \left(\frac{x_0}{2} + \sum_{n=1}^{N-1} x_n \cos\left(n\pi \frac{2k+1}{2N} \right) \right)$$

for $k = 0, 1, \ldots, (N-1)$

with $N = 6$

Parameters

- **y** – **[out]** Output vector $\bar{y}$
- **x** – **[in]** Input vector $\bar{x}$

Throws ET_LOAD_STORE
Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

void **dct8_inverse**(int32_t y[8], const int32_t x[8])

8-point 32-bit inverse DCT.

This function performs a 8-point inverse DCT (same as type-III DCT) on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result.

This operation may be safely performed in-place if x and y point to the same vector.

x and y must point to 8-byte-aligned addresses.

Operation Performed

$$y_k \leftarrow \frac{1}{N}\left(\frac{x_0}{2} + \sum_{n=1}^{N-1} x_n \cos\left(n\pi\frac{2k+1}{2N}\right)\right)$$
$$\text{for } k = 0, 1, \ldots, (N-1)$$
$$\text{with } N = 8$$

Parameters

- **y** – **[out]** Output vector $\bar{y}$
- **x** – **[in]** Input vector $\bar{x}$

Throws ET_LOAD_STORE
Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

void **dct12_inverse**(int32_t y[12], const int32_t x[12])

12-point 32-bit inverse DCT.

This function performs a 12-point inverse DCT (same as type-III DCT) on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result.

This operation may be safely performed in-place if x and y point to the same vector.

x and y must point to 8-byte-aligned addresses.

Operation Performed

$$y_k \leftarrow \frac{1}{N}\left(\frac{x_0}{2} + \sum_{n=1}^{N-1} x_n \cos\left(n\pi\frac{2k+1}{2N}\right)\right)$$
$$\text{for } k = 0, 1, \ldots, (N-1)$$
$$\text{with } N = 12$$

Parameters

- **y** – **[out]** Output vector $\bar{y}$
- **x** – **[in]** Input vector $\bar{x}$

Throws ET_LOAD_STORE
Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

void **dct16_inverse**(int32_t y[16], const int32_t x[16])

> 16-point 32-bit inverse DCT.
>
> This function performs a 16-point inverse DCT (same as type-III DCT) on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result.
>
> This operation may be safely performed in-place if x and y point to the same vector.
>
> x and y must point to 8-byte-aligned addresses.

> **Operation Performed**
>
> $$y_k \leftarrow \frac{1}{N} \left(\frac{x_0}{2} + \sum_{n=1}^{N-1} x_n \cos \left(n\pi \frac{2k+1}{2N} \right) \right)$$
> $$\text{for } k = 0, 1, \ldots, (N-1)$$
> $$\text{with } N = 16$$

> > **Parameters**
> >
> > - **y** − **[out]** Output vector $\bar{y}$
> > - **x** − **[in]** Input vector $\bar{x}$
> >
> > **Throws ET_LOAD_STORE**
> > Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

void **dct24_inverse**(int32_t y[24], const int32_t x[24])

> 24-point 32-bit inverse DCT.
>
> This function performs a 24-point inverse DCT (same as type-III DCT) on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result.
>
> This operation may be safely performed in-place if x and y point to the same vector.
>
> x and y must point to 8-byte-aligned addresses.

> **Operation Performed**
>
> $$y_k \leftarrow \frac{1}{N} \left(\frac{x_0}{2} + \sum_{n=1}^{N-1} x_n \cos \left(n\pi \frac{2k+1}{2N} \right) \right)$$
> $$\text{for } k = 0, 1, \ldots, (N-1)$$
> $$\text{with } N = 24$$

> > **Parameters**
> >
> > - **y** − **[out]** Output vector $\bar{y}$
> > - **x** − **[in]** Input vector $\bar{x}$
> >
> > **Throws ET_LOAD_STORE**
> > Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

void **dct32_inverse**(int32_t y[32], const int32_t x[32])

> 32-point 32-bit inverse DCT.
>
> This function performs a 32-point inverse DCT (same as type-III DCT) on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result.
>
> This operation may be safely performed in-place if x and y point to the same vector.
>
> x and y must point to 8-byte-aligned addresses.

Operation Performed

$$y_k \leftarrow \frac{1}{N} \left(\frac{x_0}{2} + \sum_{n=1}^{N-1} x_n \cos\left(n\pi \frac{2k+1}{2N}\right) \right)$$

for $k = 0, 1, \ldots, (N-1)$

with $N = 32$

Parameters

- **y** − **[out]** Output vector $\bar{y}$
- **x** − **[in]** Input vector $\bar{x}$

Throws ET_LOAD_STORE

Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

void **dct48_inverse**(int32_t y[48], const int32_t x[48])

48-point 32-bit inverse DCT.

This function performs a 48-point inverse DCT (same as type-III DCT) on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result.

This operation may be safely performed in-place if x and y point to the same vector.

x and y must point to 8-byte-aligned addresses.

Operation Performed

$$y_k \leftarrow \frac{1}{N} \left(\frac{x_0}{2} + \sum_{n=1}^{N-1} x_n \cos\left(n\pi \frac{2k+1}{2N}\right) \right)$$

for $k = 0, 1, \ldots, (N-1)$

with $N = 48$

Parameters

- **y** − **[out]** Output vector $\bar{y}$
- **x** − **[in]** Input vector $\bar{x}$

Throws ET_LOAD_STORE

Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

void **dct64_inverse**(int32_t y[64], const int32_t x[64])

64-point 32-bit inverse DCT.

This function performs a 64-point inverse DCT (same as type-III DCT) on input vector $\bar{x}$, and populates output vector $\bar{y}$ with the result.

This operation may be safely performed in-place if x and y point to the same vector.

x and y must point to 8-byte-aligned addresses.

Operation Performed

$$y_k \leftarrow \frac{1}{N} \left(\frac{x_0}{2} + \sum_{n=1}^{N-1} x_n \cos\left(n\pi \frac{2k+1}{2N}\right) \right)$$

for $k = 0, 1, \ldots, (N-1)$

with $N = 64$

Parameters

- **y** – **[out]** Output vector $\bar{y}$

- **x** – **[in]** Input vector $\bar{x}$

Throws ET_LOAD_STORE

Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

headroom_t **dct8x8_forward**(int8_t y[8][8], const int8_t x[8][8], const *right_shift_t* sat)

8-by-8 2D 8-bit forward DCT.

This function performs a 2-dimensional 8-by-8 type-II DCT on 8-bit input tensor $\bar{x}$ (with elements x_{rc}). Output tensor $\bar{y}$ (with elements y_{rc}) is populated with the result.

This 2D DCT is performed by first applying a 1D 8-point DCT across each row of $\bar{x}$, and then applying a 1D 8-point DCT to each column of that intermediate tensor.

The output is scaled by a factor of $2^{-\text{sat}-8}$. With $\text{sat} = 0$ this scaling is just enough to avoid any possible saturation. If saturation is considered acceptable, or known *a priori* to not be possible, negative values for sat can be used to increase precision on the output.

This operation may be safely performed in-place if x and y point to the same vector.

x and y must point to 8-byte-aligned addresses.

Operation Performed

$$y_{rc} \leftarrow \frac{4 \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} \left(x_{mn} \cos\left(c\pi \frac{2n+1}{2N}\right) \cos\left(r\pi \frac{2m+1}{2N}\right) \right)}{2^{\text{sat}+8}}$$

for $r, c \in \{0, 1, \ldots, (N-1)\}$
with $N = 8$

Parameters

- **y** – **[out]** Output vector $\bar{y}$

- **x** – **[in]** Input vector $\bar{x}$

- **sat** – **[in]** Additional output scaling exponent.

Throws ET_LOAD_STORE

Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

headroom_t **dct8x8_inverse**(int8_t y[8][8], const int8_t x[8][8], const *right_shift_t* sat)

8-by-8 2D 8-bit inverse DCT.

This function performs a 2-dimensional 8-by-8 type-III (inverse) DCT on 8-bit input tensor $\bar{x}$ (with elements x_{rc}). Output tensor $\bar{y}$ (with elements y_{rc}) is populated with the result.

This 2D DCT is performed by first applying a 1D 8-point DCT across each row of $\bar{x}$, and then applying a 1D 8-point DCT to each column of that intermediate tensor.

The output is scaled by a factor of $2^{-\text{sat}}$. With $\text{sat} = 0$ this scaling is just enough to avoid any possible saturation. If saturation is considered acceptable, or known *a priori* to not be possible, negative values for sat can be used to increase precision on the output.

This operation may be safely performed in-place if x and y point to the same vector.

x and y must point to 8-byte-aligned addresses.

Operation Performed

$$y_{rc} \leftarrow \frac{\frac{1}{N^2} \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} \left(x_{mn} \cos\left(n\pi \frac{2c+1}{2N}\right) \cos\left(m\pi \frac{2r+1}{2N}\right) \right)}{2^{\text{sat}}}$$

for $r, c \in \{0, 1, \ldots, (N-1)\}$
with $N = 8$

Parameters

- **y** − **[out]** Output vector $\bar{y}$
- **x** − **[in]** Input vector $\bar{x}$
- **sat** − **[in]** Additional output scaling exponent.

Throws ET_LOAD_STORE
Raised if x or y is not double word-aligned (See *Note: Vector Alignment*)

Variables

static const *exponent_t* **dct6_exp** = 4

Scaling exponent associated with *dct6_forward()*

Let $\bar{x}$ be the input to *dct6_forward()* and $\bar{y}$ the output. If x_exp and y_exp are the exponents associated with $\bar{x}$ and $\bar{y}$ respectively, then the following relation holds: $y\_exp = x\_exp + dct6\_exp$

static const *exponent_t* **dct8_exp** = 4

Scaling exponent associated with *dct8_forward()*

Let $\bar{x}$ be the input to *dct6_forward()* and $\bar{y}$ the output. If x_exp and y_exp are the exponents associated with $\bar{x}$ and $\bar{y}$ respectively, then the following relation holds: $y\_exp = x\_exp + dct8_exp$

static const *exponent_t* **dct12_exp** = 7

Scaling exponent associated with *dct12_forward()*

Let $\bar{x}$ be the input to *dct12_forward()* and $\bar{y}$ the output. If x_exp and y_exp are the exponents associated with $\bar{x}$ and $\bar{y}$ respectively, then the following relation holds: $y\_exp = x\_exp + dct12_exp$

static const *exponent_t* **dct16_exp** = 7

Scaling exponent associated with *dct16_forward()*

Let $\bar{x}$ be the input to *dct16_forward()* and $\bar{y}$ the output. If x_exp and y_exp are the exponents associated with $\bar{x}$ and $\bar{y}$ respectively, then the following relation holds: $y\_exp = x\_exp + dct16_exp$

static const *exponent_t* **dct24_exp** = 10

Scaling exponent associated with *dct24_forward()*

Let $\bar{x}$ be the input to *dct24_forward()* and $\bar{y}$ the output. If x_exp and y_exp are the exponents associated with $\bar{x}$ and $\bar{y}$ respectively, then the following relation holds: $y\_exp = x\_exp + dct24_exp$

static const *exponent_t* **dct32_exp** = 10

Scaling exponent associated with *dct32_forward()*

Let $\bar{x}$ be the input to *dct32_forward()* and $\bar{y}$ the output. If x_exp and y_exp are the exponents associated with $\bar{x}$ and $\bar{y}$ respectively, then the following relation holds: $y\_exp = x\_exp + dct32_exp$

static const *exponent_t* **dct48_exp** = 13

> Scaling exponent associated with *dct48_forward()*

> Let $\bar{x}$ be the input to *dct48_forward()* and $\bar{y}$ the output. If x_exp and y_exp are the exponents associated with $\bar{x}$ and $\bar{y}$ respectively, then the following relation holds: $y\_exp = x\_exp + dct48\_exp$

static const *exponent_t* **dct64_exp** = 13

> Scaling exponent associated with *dct64_forward()*

> Let $\bar{x}$ be the input to *dct64_forward()* and $\bar{y}$ the output. If x_exp and y_exp are the exponents associated with $\bar{x}$ and $\bar{y}$ respectively, then the following relation holds: $y\_exp = x\_exp + dct64\_exp$

4.4   Fast Fourier Transform API

4.4.1   FFT API Quick Reference

Table 4.9: FFT Functions - Quick Reference

Brief	Forward Function	Inverse Function
BFP FFT on single real signal	bfp_fft_forward_mono()	bfp_fft_inverse_mono()
BFP FFT on single complex signal	bfp_fft_forward_complex()	bfp_fft_inverse_complex()
BFP FFT on pair of real signals	bfp_fft_forward_stereo()	bfp_fft_inverse_stereo()
BFP spectrum packing	bfp_fft_unpack_mono()	bfp_fft_pack_mono()
Low-level decimation-in-time FFT	fft_dit_forward()	fft_dit_inverse()
Low-level decimation-in-frequency FFT	fft_dif_forward()	fft_dif_inverse()
FFT on real signal of `float`	fft_f32_forward()	fft_f32_inverse()

Functions

bfp_complex_s32_t ***bfp_fft_forward_mono**(*bfp_s32_t* *x)

Performs a forward real Discrete Fourier Transform on a real 32-bit sequence.

Performs an N-point forward real DFT on the real 32-bit BFP vector x, where N is x->length. The operation is performed in-place, resulting in an $N/2$-element complex 32-bit BFP vector.

Operation Performed

$$X[f] = \sum_{n=0}^{N-1} \left(x[n] \cdot e^{-j2\pi fn/N} \right)$$
$$\text{for } 0 \le f \le N/2$$

where $x[n]$ is the BFP vector initially represented by x, and $X[f]$ is the DFT of $x[n]$ represented by the returned pointer.

The exponent, headroom, length and data contents of x are all updated by this function, though x->data will continue to point to the same address.

x->length must be a power of 2, and must be no larger than (1<<MAX_DIT_FFT_LOG2).

This function returns a *bfp_complex_s32_t* pointer. **This points to the same address as .** This is intended as a convenience for user code.

Upon completion, the spectrum data is encoded in x->data as specified for real DFTs in Spectrum Packing. That is, x->data[f] for 1 <= f < (x->length) represent $X[f]$ for $1 \le f < (N/2)$ and x->data[0] represents $X[0] + jX[N/2]$.

Example

```
// Initialize time domain data with samples.
int32_t buffer[N] = { ... };
bfp_s32_t samples;
bfp_s32_init(&samples, buffer, 0, N, 1);
// Perform the forward DFT
{
    bfp_complex_s32_t* spectrum = bfp_fft_forward_mono(&samples);
    // `samples` should no longer be used.
    // Operate on frequency domain data using `spectrum`
    ...
    // Perform the inverse DFT to go back to time domain
    bfp_fft_inverse_mono(spectrum); // returns (bfp_s32_t*) which is the address of
↪`samples`
}
// Use `samples` again to use new time domain data.
...
```

Parameters

- **x** – **[inout]** The BFP vector $x[n]$ to be DFTed.

Returns

Address of input BFP vector x, cast as bfp_complex_s32_t*.

bfp_s32_t ***bfp_fft_inverse_mono**(*bfp_complex_s32_t* *x)

Performs an inverse real Discrete Fourier Transform on a complex 32-bit sequence.

Performs an N-point inverse real DFT on the real 32-bit BFP vector x, where N is `2*x->length`. The operation is performed in-place, resulting in an N-element real 32-bit BFP vector.

Operation Performed

$$x[n] = \sum_{f=0}^{N/2} \left(X[f] \cdot e^{j2\pi fn/N} \right)$$
$$\text{for } 0 \leq n < N$$

where $X[f]$ is the BFP vector initially represented by x, and $x[n]$ is the IDFT of $X[f]$ represented by the returned pointer.

The exponent, headroom, length and data contents of x are all updated by this function, though `x->data` will continue to point to the same address.

`x->length` must be a power of 2, and must be no larger than (`1<<(MAX_DIT_FFT_LOG2-1)`).

This function returns a *bfp_s32_t* pointer. **This points to the same address as .** This is intended as a convenience for user code.

When calling, the spectrum data must be encoded in `x->data` as specified for real DFTs in Spectrum Packing. That is, `x->data[f]` for `1 <= f < (x->length)` represent $X[f]$ for $1 \leq f < N/2$, and `x->data[0]` represents $X[0] + jX[N/2]$.

Example

```
// Initialize time domain data with samples.
int32_t buffer[N] = { ... };
bfp_s32_t samples;
bfp_s32_init(&samples, buffer, 0, N, 1);
// Perform the forward DFT
{
    bfp_complex_s32_t* spectrum = bfp_fft_forward_mono(&samples);
    // `samples` should no longer be used.
    // Operate on frequency domain data using `spectrum`
    ...
    // Perform the inverse DFT to go back to time domain
    bfp_fft_inverse_mono(spectrum); // returns (bfp_s32_t*) which is the address of
↪`samples`
}
// Use `samples` again to use new time domain data.
...
```

Parameters

- **x** − **[inout]** The BFP vector $X[f]$ to be IDFTed.

Returns

Address of input BFP vector x, cast as `bfp_s32_t*`.

void **bfp_fft_forward_complex**(*bfp_complex_s32_t* *x)

Performs a forward complex Discrete Fourier Transform on a complex 32-bit sequence.

Performs an N-point forward complex DFT on the complex 32-bit BFP vector x, where N is `x->length`. The operation is performed in-place.

Operation Performed

$$X[f] = \sum_{n=0}^{N-1} \left(x[n] \cdot e^{-j2\pi fn/N} \right)$$
$$\text{for } 0 \leq f < N$$

where $x[n]$ is the BFP vector initially represented by x, and $X[f]$ is the DFT of $x[n]$, also represented by x upon completion.

The exponent, headroom and data contents of x are updated by this function. `x->data` will continue to point to the same address.

`x->length` (N) must be a power of 2, and must be no larger than (`1<<MAX_DIT_FFT_LOG2`).

Upon completion, the spectrum data is encoded in x as specified in Spectrum Packing. That is, `x->data[f]` for `0 <= f < (x->length)` represent $X[f]$ for $0 \leq f < N$.

Example

```
// Initialize complex time domain data with samples.
complex_s32_t buffer[N] = { ... };
bfp_complex_s32_t vector;
bfp_complex_s32_init(&vector, buffer, 0, N, 1);
// Perform the forward DFT
bfp_fft_forward_mono(&vector);
// Operate on frequency domain data
...
// Perform the inverse DFT to go back to time domain
bfp_fft_inverse_mono(&vector);
// `vector` contains (complex) time-domain data again
...
```

Parameters

- **x** − **[inout]** The BFP vector $x[n]$ to be DFTed.

void **bfp_fft_inverse_complex**(*bfp_complex_s32_t* *x)

Performs an inverse complex Discrete Fourier Transform on a complex 32-bit sequence.

Performs an N-point inverse complex DFT on the complex 32-bit BFP vector x, where N is `x->length`. The operation is performed in-place.

Operation Performed

$$x[n] = \sum_{f=0}^{N-1} \left(X[f] \cdot e^{j2\pi fn/N} \right)$$
$$\text{for } 0 \leq f < N$$

where $X[f]$ is the BFP vector initially represented by x, and $x[n]$ is the DFT of $X[f]$, also represented by x upon completion.

The exponent, headroom and data contents of x are updated by this function. `x->data` will continue to point to the same address.

`x->length` must be a power of 2, and must be no larger than (`1<<MAX_DIT_FFT_LOG2`).

The data initially encoded in x are interpreted as specified in Spectrum Packing. That is, `x->data[f]` for `0 <= f < (x->length)` represent $X[f]$ for $0 \leq f < N$.

Example

```
// Initialize complex time domain data with samples.
complex_s32_t buffer[N] = { ... };
bfp_complex_s32_t vector;
bfp_complex_s32_init(&vector, buffer, 0, N, 1);
// Perform the forward DFT
bfp_fft_forward_mono(&vector);
// Operate on frequency domain data
...
// Perform the inverse DFT to go back to time domain
bfp_fft_inverse_mono(&vector);
// `vector` contains (complex) time-domain data again
...
```

Parameters

- **x** – **[inout]** The BFP vector $x[n]$ to be IDFTed.

void **bfp_fft_forward_stereo**(*bfp_s32_t* *a, *bfp_s32_t* *b, *complex_s32_t* scratch[])

Performs a forward real Discrete Fourier Transform on a pair of real 32-bit sequences.

Performs an N-point forward real DFT on the real 32-bit BFP vectors $\bar{a}$ and $\bar{b}$, where N is a->length (which must equal b->length). The resulting spectra, $\bar{A}$ and $\bar{B}$, are placed in a and b. Each spectrum is a $N/2$-element complex 32-bit BFP vectors. To access the spectrum, the pointers a and b should be cast to bfp_complex_s32_t* following a call to this function.

Operation Performed

$$A[f] = \sum_{n=0}^{N-1} \left(a[n] \cdot e^{-j2\pi fn/N} \right) \text{ for } 0 \le f \le N/2$$

$$B[f] = \sum_{n=0}^{N-1} \left(b[n] \cdot e^{-j2\pi fn/N} \right) \text{ for } 0 \le f \le N/2$$

where $a[n]$ and $b[n]$ are the two time-domain sequences represented by input BFP vectors a and b, and $A[f]$ and $B[f]$ are the DFT of $a[n]$ and $b[n]$ respectively.

a->length (N) must be equal to b->length, must be a power of 2, and must be no larger than(1«MAX_DIT_FFT_LOG2)'.

The parameters and are used as both inputs and outputs. To access the result of the FFT, a and b should be cast to *bfp_complex_s32_t**. The structs' metadata (e.g. exp, hr, length) are updated by this function to reflect this change of interpretation. The *bfp_s32_t* references should be considered corrupted after this call (at least until *bfp_fft_inverse_stereo()* is called).

The spectrum data is encoded in a->data and b->data as specified for real DFTs in Spectrum Packing. That is, a->data[f] for 1 <= f < (a->length) represent $A[f]$ for $1 \le f < (N/2)$ and a->data[0] represents $A[0] + jA[N/2]$. Likewise for the encoding of b->data.

This function requires a scratch buffer large enough to contain N *complex_s32_t* elements.

Deprecated:

Example

```
// Initialize time domain data with samples.
int32_t bufferA[N] = { ... };
int32_t bufferB[N] = { ... };
complex_s32_t scratch[N]; // scratch buffer -- contents don't matter
bfp_s32_t channel_A, channel_B;
bfp_s32_init(&channel_A, buffer, 0, N, 1);
bfp_s32_init(&channel_B, buffer, 0, N, 1);

// Perform the forward DFT
bfp_fft_forward_stereo(&channel_A, &channel_B, scratch);

// channel_A and channel_B should now be considered clobbered as the structs are now
// effectively bfp_complex_s32_t
bfp_complex_s32_t* chanA = (bfp_complex_s32_t*) &channel_A;
bfp_complex_s32_t* chanB = (bfp_complex_s32_t*) &channel_B;

// Operate on frequency domain data using `chanA` and `chanB`
...
// Perform the inverse DFT to go back to time domain
bfp_fft_inverse_stereo(&chanA, &chanB, scratch);

// Use channel_A and channel_B again to use new time domain data.
...
```

// Suppress this from generated documentation for the time being //

Note:

Use of this function is not currently recommended. It functions correctly, but a recent change in this library's API (namely, dropping support for channel-pair vectors) means this function is no more computationally efficient than calling *bfp_fft_forward_mono()* on each input vector separately. Additionally, this function currently requires a scratch buffer, whereas the mono FFT does not.

Parameters

- **a** − **[inout]** [Input] Time-domain BFP vector $\bar{a}$. [Output] Frequency domain BFP vector $\bar{A}$

- **b** − **[inout]** [Input] Time-domain BFP vector $\bar{b}$. [Output] Frequency domain BFP vector $\bar{B}$

- **scratch** − Scratch buffer of at least `a->length` *complex_s32_t* elements

void **bfp_fft_inverse_stereo**(*bfp_complex_s32_t* *A_fft, *bfp_complex_s32_t* *B_fft, *complex_s32_t* scratch[])

Performs an inverse real Discrete Fourier Transform on a pair of complex 32-bit sequences.

Performs an N-point inverse real DFT on the 32-bit complex BFP vectors $\bar{A}$ and $\bar{B}$ (A_fft and B_fft respectively), where N is A_fft->length . The resulting real signals, $\bar{a}$ and $\bar{b}$, are placed in A_fft and B_fft. Each time-domain result is a $N/2$-element real 32-bit BFP vectors. To access the spectrum, the pointers A_fft and B_fft should be cast to bfp_s32_t* following a call to this function.

Operation Performed

$$a[n] = \sum_{f=0}^{N/2-1} \left(A[f] \cdot e^{j2\pi fn/N} \right) \text{ for } 0 \leq n < N$$

$$b[n] = \sum_{f=0}^{N/2-1} \left(B[f] \cdot e^{j2\pi fn/N} \right) \text{ for } 0 \leq n < N$$

where $A[f]$ and $B[f]$ are the frequency spectra represented by BFP vectors `A_fft` and `B_fft`, and $a[n]$ and $b[n]$ are the IDFT of $A[f]$ and $B[f]$.

`A_fft->length` (N) must be a power of 2, and must be no larger than (`1<<(MAX_DIT_FFT_LOG2-1)`).

The parameters and are used as both inputs and outputs. To access the result of the IFFT, `A_fft` and `B_fft` should be cast to *`bfp_s32_t`**. The structs' metadata (e.g. `exp`, `hr`, `length`) are updated by this function to reflect this change of interpretation. The *`bfp_complex_s32_t`* references should be considered corrupted after this call.

The spectrum data encoded in `A_fft->data` and `A_fft->data` are interpreted as specified for real DFTs in Spectrum Packing. That is, `A_fft->data[f]` for `1 <= f < (a->length)` represent $A[f]$ for $1 \leq f < (N/2)$ and `A_fft->data[0]` represents $A[0] + jA[N/2]$. Likewise for the encoding of `B_fft->data`.

This function requires a scratch buffer large enough to contain $2N$ *`complex_s32_t`* elements.

Deprecated:

Example

```
// Initialize time domain data with samples.
int32_t bufferA[N] = { ... };
int32_t bufferB[N] = { ... };
complex_s32_t scratch[N]; // scratch buffer -- contents don't matter
bfp_s32_t channel_A, channel_B;
bfp_s32_init(&channel_A, buffer, 0, N, 1);
bfp_s32_init(&channel_B, buffer, 0, N, 1);

// Perform the forward DFT
bfp_fft_forward_stereo(&channel_A, &channel_B, scratch);

// channel_A and channel_B should now be considered clobbered as the structs are now
// effectively bfp_complex_s32_t
bfp_complex_s32_t* chanA = (bfp_complex_s32_t*) &channel_A;
bfp_complex_s32_t* chanB = (bfp_complex_s32_t*) &channel_B;

// Operate on frequency domain data using `chanA` and `chanB`
...
// Perform the inverse DFT to go back to time domain
bfp_fft_inverse_stereo(&chanA, &chanB, scratch);

// Use channel_A and channel_B again to use new time domain data.
...
```

// Suppress this from generated documentation for the time being //

Note:

Use of this function is not currently recommended. It functions correctly, but a recent change in this library's API (namely, dropping support for channel-pair vectors) means this function is no more

computationally efficient than calling `bfp_fft_forward_mono()` on each input vector separately. Additionally, this function currently requires a scratch buffer, whereas the mono FFT does not.

Parameters

- **A_fft** – **[inout]** [Input] Freq-domain BFP vector $\bar{A}$. [Output] Time domain BFP vector $\bar{b}$

- **B_fft** – **[inout]** [Input] Freq-domain BFP vector $\bar{b}$. [Output] Time domain BFP vector $\bar{b}$

- **scratch** – Scratch buffer of at least `2*A_fft->length` `complex_s32_t` elements

void **bfp_fft_unpack_mono**(*bfp_complex_s32_t* *x)

Unpack the spectrum resulting from *bfp_fft_forward_mono()*.

The DFT of a real signal is periodic with period FFT_N (the FFT length) and has a complex conjugate symmetry about index 0. These two properties guarantee that the imaginary part of both the DC component (index 0) and the Nyquist component (index FFT_N/2) of the spectrum are zero. To compute the forward FFT in-place, *bfp_fft_forward_mono()* packs the real part of the Nyquist rate component of the output spectrum into the imaginary part of the DC component.

This may be undesirable when operating on the signal's complex spectrum. Use this function to unpack the Nyquist component. This function will also adjust the BFP vector's length to reflect this unpacking.

NOTE: If you intend to unpack the spectrum using this function, the buffer for the time-domain BFP vector must have length `FFT_N+2`, rather than `FFT_N` (`int32_t` elements), but these should NOT be reflected in the time-domain BFP vector's `length` field.

Operation Performed

$$\begin{aligned}
Rex_{N/2} &\leftarrow Imx_0 \\
Imx_0 &\leftarrow 0 \\
Imx_{N/2} &\leftarrow 0 \\
x.length &\leftarrow x.length + 1
\end{aligned}$$

NOTE: Before *bfp_fft_inverse_mono()* may be applied, *bfp_fft_pack_mono()* must be called, as the inverse FFT expects the data to be packed.

See also:

bfp_fft_forward_mono, *bfp_fft_pack_mono*

Parameters

- **x** – **[inout]** The spectrum to be unpacked

void **bfp_fft_pack_mono**(*bfp_complex_s32_t* *x)

Pack the spectrum resulting from *bfp_fft_unpack_mono()*.

This function applies the reverse process of *bfp_fft_unpack_mono()*, to prepare it for an inverse FFT using *bfp_fft_inverse_mono()*.

See also:

bfp_fft_inverse_mono, *bfp_fft_unpack_mono*

Parameters

- **x** – **[inout]** The spectrum to be packed

void **fft_dit_forward**(*complex_s32_t* x[], const unsigned N, *headroom_t* *hr, *exponent_t* *exp)

Compute a forward DFT using the decimation-in-time FFT algorithm.

This function computes the N-point forward DFT of a complex input signal using the decimation-in-time FFT algorithm. The result is computed in-place.

Operation Performed

$$X[f] = \frac{1}{2^\alpha} \sum_{n=0}^{N-1} \left(x[n] \cdot e^{-j2\pi fn/N} \right)$$
$$\text{for } 0 \le f < N$$

x[] is interpreted to be a block floating-point vector with shared exponent *exp and with *hr bits of headroom initially in x[]. During computation, this function monitors the headroom of the data and compensates to avoid overflows and underflows by bit-shifting the data up or down as appropriate. In the equation above, α represents the (net) number of bits that the data was right-shifted by.

Upon completion, *hr is updated with the final headroom in x[], and the exponent *exp is incremented by α.

Note: In order to guarantee that saturation will not occur, x[] must have an *initial* headroom of at least 2 bits.

Parameters

- **x** – **[inout]** The N-element complex input vector to be transformed.
- **N** – **[in]** The size of the DFT to be performed.
- **hr** – **[inout]** Pointer to the initial headroom in x[].
- **exp** – **[inout]** Pointer to the initial exponent associated with x[].

Throws ET_LOAD_STORE
Raised if x is not word-aligned (See *Note: Vector Alignment*)

void **fft_dit_inverse**(*complex_s32_t* x[], const unsigned N, *headroom_t* *hr, *exponent_t* *exp)

Compute an inverse DFT using the decimation-in-time IFFT algorithm.

This function computes the N-point inverse DFT of a complex spectrum using the decimation-in-time IFFT algorithm. The result is computed in-place.

Operation Performed

$$x[n] = \frac{1}{2^\alpha} \sum_{f=0}^{N-1} \left(X[f] \cdot e^{j2\pi fn/N} \right)$$
$$\text{for } 0 \le n < N$$

x[] is interpreted to be a block floating-point vector with shared exponent *exp and with *hr bits of headroom initially in x[]. During computation, this function monitors the headroom of the data and compensates to avoid overflows and underflows by bit-shifting the data up or down as appropriate. In the equation above, α represents the (net) number of bits that the data was right-shifted by.

Upon completion, `*hr` is updated with the final headroom in `x[]`, and the exponent `*exp` is incremented by α.

Note: In order to guarantee that saturation will not occur, `x[]` must have an *initial* headroom of at least 2 bits.

Parameters

- **x** – **[inout]** The `N`-element complex input vector to be transformed.
- **N** – **[in]** The size of the inverse DFT to be performed.
- **hr** – **[inout]** Pointer to the initial headroom in `x[]`.
- **exp** – **[inout]** Pointer to the initial exponent associated with `x[]`.

Throws ET_LOAD_STORE
Raised if `x` is not word-aligned (See *Note: Vector Alignment*)

void **fft_dif_forward**(*complex_s32_t* x[], const unsigned N, *headroom_t* *hr, *exponent_t* *exp)

Compute a forward DFT using the decimation-in-frequency FFT algorithm.

This function computes the `N`-point forward DFT of a complex input signal using the decimation-in-frequency FFT algorithm. The result is computed in-place.

Operation Performed

$$X[f] = \frac{1}{2^\alpha} \sum_{n=0}^{N-1} \left(x[n] \cdot e^{-j2\pi fn/N} \right)$$
$$\text{for } 0 \leq f < N$$

`x[]` is interpreted to be a block floating-point vector with shared exponent `*exp` and with `*hr` bits of headroom initially in `x[]`. During computation, this function monitors the headroom of the data and compensates to avoid overflows and underflows by bit-shifting the data up or down as appropriate. In the equation above, α represents the (net) number of bits that the data was right-shifted by.

Upon completion, `*hr` is updated with the final headroom in `x[]`, and the exponent `*exp` is incremented by α.

Note: In order to guarantee that saturation will not occur, `x[]` must have an *initial* headroom of at least 2 bits.

Parameters

- **x** – **[inout]** The `N`-element complex input vector to be transformed.
- **N** – **[in]** The size of the DFT to be performed.
- **hr** – **[inout]** Pointer to the initial headroom in `x[]`.
- **exp** – **[inout]** Pointer to the initial exponent associated with `x[]`.

Throws ET_LOAD_STORE
Raised if `x` is not word-aligned (See *Note: Vector Alignment*)

void **fft_dif_inverse**(*complex_s32_t* x[], const unsigned N, *headroom_t* *hr, *exponent_t* *exp)

Compute an inverse DFT using the decimation-in-frequency IFFT algorithm.

This function computes the `N`-point inverse DFT of a complex spectrum using the decimation-in-frequency IFFT algorithm. The result is computed in-place.

Operation Performed

$$x[n] = \frac{1}{2^\alpha} \sum_{f=0}^{N-1} \left(X[f] \cdot e^{j2\pi fn/N} \right)$$
$$\text{for } 0 \leq n < N$$

`x[]` is interpreted to be a block floating-point vector with shared exponent `*exp` and with `*hr` bits of headroom initially in `x[]`. During computation, this function monitors the headroom of the data and compensates to avoid overflows and underflows by bit-shifting the data up or down as appropriate. In the equation above, α represents the (net) number of bits that the data was right-shifted by.

Upon completion, `*hr` is updated with the final headroom in `x[]`, and the exponent `*exp` is incremented by α.

Note: In order to guarantee that saturation will not occur, `x[]` must have an *initial* headroom of at least 2 bits.

Parameters

- **x** – **[inout]** The `N`-element complex input vector to be transformed.
- **N** – **[in]** The size of the inverse DFT to be performed.
- **hr** – **[inout]** Pointer to the initial headroom in `x[]`.
- **exp** – **[inout]** Pointer to the initial exponent associated with `x[]`.

Throws ET_LOAD_STORE
Raised if `x` is not word-aligned (See *Note: Vector Alignment*)

4.5 Filtering API

Table 4.10: Filtering API - Quick Reference

Filter	Function	Brief
32-bit FIR	`filter_fir_s32_init()`	Initialize filter
32-bit FIR	`filter_fir_s32_add_sample()`	Add sample (without computing output)
32-bit FIR	`filter_fir_s32()`	Process next sample
16-bit FIR	`filter_fir_s16_init()`	Initialize filter
16-bit FIR	`filter_fir_s16_add_sample()`	Add sample (without computing output)
16-bit FIR	`filter_fir_s16()`	Process next sample
32-bit Biquad	`filter_biquad_s32()`	Process next sample (single block)
32-bit Biquad	`filter_biquads_s32()`	Process next sample (multi block)

group **filter_api**

Functions

void **filter_fir_s32_init**(*filter_fir_s32_t* *filter, int32_t *sample_buffer, const unsigned tap_count, const int32_t *coefficients, const *right_shift_t* shift)

Initialize a 32-bit FIR filter.

Before *filter_fir_s32()* or *filter_fir_s32_add_sample()* can be used on a filter it must be initialized with a call to this function.

`sample_buffer` and `coefficients` must be at least `4 * tap_count` bytes long, and aligned to a 4-byte (word) boundary.

See `filter_fir_s32_t` for more information about 32-bit FIR filters and their operation.

See also:

filter_fir_s32_t

Parameters

- **filter** – **[out]** Filter struct to be initialized
- **sample_buffer** – **[in]** Buffer used by the filter to contain state information. Must be at least `tap_count` elements long
- **tap_count** – **[in]** Order of the FIR filter; number of filter taps
- **coefficients** – **[in]** Array containing filter coefficients.
- **shift** – **[in]** Unsigned arithmetic right-shift applied to accumulator to get filter output sample

void **filter_fir_s32_add_sample**(*filter_fir_s32_t* *filter, const int32_t new_sample)

Add a new input sample to a 32-bit FIR filter without processing an output sample.

This function adds a new input sample to `filter`'s state without computing a new output sample. This is a constant- time operation and can be used to quickly pre-load a filter with sample data.

See `filter_fir_s32_t` for more information about FIR filters and their operation.

See also:

filter_fir_s32_t

Parameters

- **filter** – **[inout]** Filter struct to have the sample added
- **new_sample** – **[in]** Sample to be added to `filter`'s history

int32_t **filter_fir_s32**(*filter_fir_s32_t* *filter, const int32_t new_sample)

This function implements a Finite Impulse Response (FIR) filter.

The new input sample `new_sample` is added to this filter's state, and a new output sample is computed and returned as specified in `filter_fir_s32_t`.

With a large number of filter taps, this function takes approximately 3 thread cycles per 8 filter taps.

> **Parameters**
> - **filter** – **[inout]** Filter to be processed
> - **new_sample** – **[in]** New input sample to be processed by `filter`
>
> **Returns**
> Next filtered output sample

void **filter_fir_s16_init**(*filter_fir_s16_t* *filter, int16_t *sample_buffer, const unsigned tap_count, const int16_t *coefficients, const *right_shift_t* shift)

Initialize a 16-bit FIR filter.

Before *filter_fir_s16()* or *filter_fir_s16_add_sample()* can be used on a filter it must be initialized with a call to this function.

`sample_buffer` and `coefficients` must be at least `2 * tap_count` bytes long, and aligned to a 4-byte (word) boundary.

See `filter_fir_s16_t` for more information about 16-bit FIR filters and their operation.

> **Parameters**
> - **filter** – **[out]** Filter struct to be initialized
> - **sample_buffer** – **[in]** Buffer used by the filter to contain state information. Must be at least `tap_count` elements long
> - **tap_count** – **[in]** Order of the FIR filter; number of filter taps
> - **coefficients** – **[in]** Array containing filter coefficients
> - **shift** – **[in]** Unsigned arithmetic right-shift applied to accumulator to get filter output sample

void **filter_fir_s16_add_sample**(*filter_fir_s16_t* *filter, const int16_t new_sample)

Add a new input sample to a 16-bit FIR filter without processing an output sample.

This function adds a new input sample to `filter`'s state without computing a new output sample.

See `filter_fir_s16_t` for more information about FIR filters and their operation.

> **Parameters**
> - **filter** – **[inout]** Filter struct to have the sample added
> - **new_sample** – **[in]** Sample to be added to `filter`'s history

int16_t **filter_fir_s16**(*filter_fir_s16_t* *filter, const int16_t new_sample)

This function implements a Finite Impulse Response (FIR) filter.

The new input sample `new_sample` is added to this filter's state, and a new output sample is computed and returned as specified in *filter_fir_s16_t*.

With a large number of filter taps, this function takes approximately 3 thread cycles per 16 filter taps.

See also:

filter_fir_s16_t

Parameters

- **`filter`** – **[inout]** Filter to be processed
- **`new_sample`** – **[in]** New input sample to be processed by `filter`

Returns

Next filtered output sample

int32_t **filter_biquad_s32**(*filter_biquad_s32_t* *filter, const int32_t new_sample)

This function implements a 32-bit Biquad filter.

The new input sample `new_sample` is added to this filter's state, and a new output sample is computed and returned as specified in *filter_biquad_s32_t*.

This function processes a single filter block containing (up to) 8 biquad filter sections. For biquad filters containing 2 or more filter blocks (more than 8 biquad filter sections), see *filter_biquads_s32()*.

See also:

filter_biquad_s32_t, *filter_biquads_s32*

Parameters

- **`filter`** – **[inout]** Filter to be processed
- **`new_sample`** – **[in]** New input sample to be processed by `filter`

Returns

Next filtered output sample

int32_t **filter_biquads_s32**(*filter_biquad_s32_t* biquads[], const unsigned block_count, const int32_t new_sample)

This function implements a 32-bit Biquad filter.

The new input sample `new_sample` is added to this filter's state, and a new output sample is computed and returned as specified in *filter_biquad_s32_t*.

This function processes one or more filter blocks, with each block containing up to 8 biquad filter sections.

See also:

filter_biquad_s32_t, *filter_biquad_s32*

Parameters

- **`biquads`** – **[inout]** Filter blocks to be processed

- **block_count** – **[in]** Number of filter blocks in `biquads`

- **new_sample** – **[in]** New input sample to be processed by `filter`

Returns

Next filtered output sample

struct **filter_fir_s32_t**

#include <filter.h> 32-bit Discrete-Time Finite Impulse Response (FIR) Filter

Todo:

Move most of this information out to higher-level documentation

Filter Model

This struct represents an N-tap 32-bit discrete-time FIR Filter.

At each time step, the FIR filter consumes a single 32-bit input sample and produces a single 32-bit output sample.

To process a new input sample and compute a new output sample, use *filter_fir_s32()*. To add a new input sample to the filter without computing a new output sample, use *filter_fir_s32_add_sample()*.

An N-tap FIR filter contains N 32-bit cofficients (pointed to by `coef`) and N words of state data (pointed to by `state`. The state data is a vector of the N most recent input samples. When processing a new input sample at time step t, $x[t]$ is the new input sample, $x[t-1]$ is the previous input sample, and so on, up to $x[t-(N-1)]$, which is the oldest input considered when computing the new output sample (see note 1 below). The coefficients form a vector $b[]$, where $b[k]$ is the coefficient by which the kth oldest input sample is multiplied. There is an additional parameter `shift` which scales the output as described below. Both the coefficients and `shift` are considered to be constants which do not change after initialization (although nothing should break if they are changed to new valid values).

At time step t, the output sample $y[t]$ is computed based on the inner product (i.e. sum of element-wise products) of the coefficients and state data as follows (a more detailed description is below):

```
acc = x[t-0] * b[0] + x[t-1] * b[1] + x[t-2] * b[2] + ... + x[t-(N-1)] * b[N-1]
y[t] = acc >> shift
```

Importantly, all three of the operators above (addition, multiplication and the rightwards bitshift) have slightly ideosyncratic meanings.

The products have a built-in rounding arithmetic right-shift of 30 bits, where ties round toward positive infinity. This is a hardware feature which allows for longer filters (larger N) without sacrificing coefficient precision. These element-wise products accumulate into 8 40-bit accumulators saturate the sums at symmetric 40-bit bounds (see Symmetrically Saturating Arithmetic). The order in which the taps are accumulated is unspecified (see note 2 below).

After each tap has been accumulated, the 8 accumulators are then added together to get a 64-bit penultimate result (with 43 useful bits). Finally, an unsigned rounding arithmetic right-shift of `shift` bits is applied to the 64-bit sum, and the final result is saturated to the symmetric 32-bit range (`-INT32_MAX` to `INT32_MAX` inclusive).

Below is a more detailed description of the operations performed (not including the saturation logic applied by the accumulators).

$$y[t] = sat_{32}\left(round\left(\left(\sum_{k=0}^{N-1} round(x[t-k] \cdot b[k] \cdot 2^{-30})\right) \cdot 2^{-shift}\right)\right)$$

where $sat_{32}()$ saturates to $\pm(2^{31} - 1)$

and $round()$ rounds to the nearest integer, with ties rounding towards $+\infty$

Operations

Initialize: A `filter_fir_s32_t` filter is initialized with a call to *filter_fir_s32_init()*. The caller supplies information about the filter, including the number of taps and pointers the coefficients and a state buffer. It is typically recommended that the state buffer be cleared to all `0`s before initializing.

Add Sample: To add a new input sample without computing a new output sample, use *filter_fir_s32_add_sample()*. This is a constant-time operation which does not depend on the number of filter taps. This may be useful in some situations, for example, to quickly pre-load the filter's state buffer with multiple samples, without incurring the cost of computing an output with each added sample.

Process Sample: To process a new input sample and produce a new output sample, use *filter_fir_s32()*.

Fields

After initialization via *filter_fir_s32_init()*, the contents of the `filter_fir_s32_t` struct are considered to be opaque, and may change between major versions. In general, user code should not need to access its members.

`num_taps` is the order of the filter, or the number of taps. It is also the (minimum) size of the buffers to which `coef` and `state` point, in elements (where each element is 4 bytes). The time required to process an input sample and produce an output sample is approximately linear in `num_taps` (see Performance below).

`head` is the index into `state` at which the next sample will be added.

`shift` is the unsigned arithmetic rounding saturating right-shift applied to internal accumulator to get a final output.

`coef` is a pointer to a buffer (supplied by the user at initialization) containing the tap coefficients. The coefficients are stored in forward order, with lower indices corresponding to newer samples. `coef[0]`, then, corresponds to `b[0]`, `coef[1]` to `b[1]`, and so on. None of the functions which operate on `filter_fir_s32_t` structs in this library will modify the contents of the buffer to which `coef` points. This buffer must be at least `num_taps` words long.

`state` is a pointer to a buffer (supplied by the user at initialization) containing the state data — a history of the `num_taps` most recent input samples. `state` is used in a circular fashion with `head` indicating the index at which the next sample will be inserted.

Performance

More work remains to fully characterize the time performance of this FIR filter, but asymptotically (i.e. with a large number of filter taps) processing a new input sample to produce a new output sample takes approximately 3 thread cycles per 8 filter taps.

That assumes that both the coefficients (pointed to by `coef`) and state buffer (pointed to by `state`) are stored directly in SRAM.

Coefficient Scaling

Suppose you're starting with a floating-point FIR filter model with coefficients `B[k]` which operates on a sequence of 32-bit integer input samples `x[t]` to get a result `Y[t]` where

```
Y[t] = x[t-0] * B[0] + x[t-1] * B[1] + ... + x[t-(N-1)] * B[N-1]
```

Because of the 30-bit right-shift and the right-shift of the final accumulator by `shift` bits, the coefficients `b[k]` to use with this library can be thought of as fixed-point values with `30 + shift` fractional bits.

The floating-point coefficients `B[k]` can then be naively converted to fixed-point coefficients `b[k]`

```
shift = 0
b[k] = (int32_t) round(ldexp(B[k], 30)
```

After this, any further doubling of the coefficients can be compensated for without changing the overall gain by incrementing `shift`.

To maximize precision, you'll typically want `shift` to be as large as possible while in the worst case to be considered neither saturates the internal accumulator (which, for safety, should generally be assumed to be 42 bits), nor saturates the final 32-bit output when `shift` is applied.

The details of this depend on various details, such as your filter's gain and the statistics of the sequence `x[t]` (e.g. any headroom `x[t]` is known *a priori* to have).

Filter Conversion

This library includes a python script which converts existing floating-point FIR filter coefficients into a suitable representation and generates code for easily initializing and executing the filter. See *Note: Digital Filter Conversion* for more.

Usage Example

```
#define N        256                  // Tap count
#define B_VAL    ldexp(1.0/N, 30+7)   // Value for (all) coefficients

const int32_t b[TAPS] =               // The filter coefficients
{ B_VAL, B_VAL, B_VAL, ..., B_VAL };
const right_shift_t shift = 7;        // The (unsigned) right-shift applied to the
→final accumulator
int32_t state_buff[TAPS] = { 0 };     // Filter state buffer, initialized to 0's
filter_fir_s32_t filter;             // The filter struct

#define SAMPLE_COUNT    1024
int32_t x[SAMPLE_COUNT] = { ... };    // Some sequence of input samples
```

```
// Initialize
filter_fir_s32_init(&filter, state_buff, N, b, shift);

// Just add the first 64 without processing output samples. (not necessary)
for(unsigned i = 0; i < 64; i++)
    filter_fir_s32_add_sample(&filter, x[i]);

// Process the rest, generating a sequence of filtered output samples
int32_t y[SAMPLE_COUNT] = { 0 };        //Output samples (first 64 never get updated
↪here)
for(unsigned i = 64; i < SAMPLE_COUNT; i++)
    y[i] = filter_fir_s32(&filter, x[i]);

// Do something with output sequence
...
```

This example creates a simple 256-tap filter which averages the most recent 256 samples.

Each `b[k]` is 2^{29}, and the final accumulator is right-shifted 7 bits. In the worst case, all input samples are -2^{31}. In that case, the final accumulator value is $256 \cdot (2^{29} \cdot -2^{31} \cdot 2^{-30}) = -2^{38}$, well below the saturation limit of the accumulator. After `shift` is applied, that becomes $-2^{38} \cdot 2^{-7} = -2^{31}$. Finally, the 32-bit symmetric saturation logic is applied, making the final output value $-2^{31} + 1$.

Notes

a. `state` is a circular buffer, and so the index of `x[t]` within `state` changes with each input sample. The `state` field of this struct is considered to be opaque — its exact usage may change between versions.

b. Ordinarily integer sums are associative, so the order in which elements are added added does not affect the final result. The sum that the FIR filters use, however, is saturating, with the saturation logic being applied throughout the sum. This saturation is a hard non-linearity and is *not* associative. The details of exactly when each tap is accumulated and into which accumulator are complicated and subject to change. It is best to construct a filter such that no ordering of the taps will saturate the accumulators.

See also:

filter_fir_s32_init, *filter_fir_s32_add_sample*, *filter_fir_s32*

struct **filter_fir_s16_t**

#include <filter.h> 16-bit Discrete-Time Finite Impulse Response (FIR) Filter

Filter Model

This struct represents an N-tap 16-bit discrete-time FIR Filter.

At each time step, the FIR filter consumes a single 16-bit input sample and produces a single 16-bit output sample.

To process a new input sample and compute a new output sample, use *filter_fir_s16()*. To add a new input sample to the filter without computing a new output sample, use *filter_fir_s16_add_sample()*.

An `N`-tap FIR filter contains `N` 16-bit cofficients (pointed to by `coef`) and `N` `int16_t`s of state data (pointed to by `state`. The state data is a vector of the `N` most recent input samples. When processing a new input sample at time step `t`, `x[t]` is the new input sample, `x[t-1]` is the previous input sample, and so on, up to `x[t-(N-1)]`, which is the oldest input considered when computing the new output sample (see note 1 below). The coefficients form a vector `b[]`, where `b[k]` is the coefficient by which the `k`th oldest input sample is multiplied. There is an additional parameter `shift` which scales the output as described below. Both the coefficients and `shift` are considered to be constants which do not change after initialization (although nothing should break if they are changed to new valid values).

At time step `t`, the output sample `y[t]` is computed based on the inner product (i.e. sum of element-wise products) of the coefficients and state data as follows (a more detailed description is below):

```
acc = x[t-0] * b[0] + x[t-1] * b[1] + x[t-2] * b[2] + ... + x[t-(N-1)] * b[N-1]
y[t] = acc >> shift
```

Unlike the 32-bit FIR filters (see `filter_fir_s16_t`), the products `x[t-k] * b[k]` are the raw 32-bit products of the 16-bit elements. These element-wise products accumulate into a 32-bit accumulator which saturates the sums at symmetric 32-bit bounds (see Symmetrically Saturating Arithmetic).

After all taps have been accumulated, a rounding arithmetic right-shift of `shift` bits is applied to the 64-bit sum, and the final result is saturated to the symmetric 16-bit range (the open interval $(-2^{15}, 2^{15})$).

Below is a more detailed description of the operations performed (not including the saturation logic applied by the accumulators).

$$y[t] = sat_{16}\left(round\left(\left(\sum_{k=0}^{N-1} round(x[t-k] \cdot b[k])\right) \cdot 2^{-shift}\right)\right)$$

where $sat_{32}()$ saturates to $\pm(2^{15} - 1)$
and $round()$ rounds to the nearest integer, with ties rounding towards $+\infty$

Operations

Initialize: A `filter_fir_s16_t` filter is initialized with a call to *filter_fir_s16_init()*. The caller supplies information about the filter, including the number of taps and pointers the coefficients and a state buffer. It is typically recommended that the state buffer be cleared to all `0`s before initializing.

Add Sample: To add a new input sample without computing a new output sample, use *filter_fir_s16_add_sample()*. Unlike *filter_fir_s32_add_sample()*, this is not a constant-time operation, and does depend on the number of filter taps. Nevertheless, this is faster than computing output samples, and may be useful in some situations, for example, to moer quickly pre-load the filter's state buffer with multiple samples, without incurring the cost of computing an output with each added sample.

Process Sample: To process a new input sample and produce a new output sample, use *filter_fir_s16()*.

Fields

After initialization via *filter_fir_s16_init()*, the contents of the `filter_fir_s16_t` struct are considered to be opaque, and may change between major versions. In general, user code should not need to access its members.

`num_taps` is the order of the filter, or the number of taps. It is also the (minimum) size of the buffers to which `coef` and `state` point, in elements (where each element is 2 bytes). The time required to process an input sample and produce an output sample is approximately linear in `num_taps` (see Performance below).

`shift` is the unsigned arithmetic rounding saturating right-shift applied to internal accumulator to get a final output.

`coef` is a pointer to a buffer (supplied by the user at initialization) containing the tap coefficients. The coefficients are stored in forward order, with lower indices corresponding to newer samples. `coef[0]`, then, corresponds to `b[0]`, `coef[1]` to `b[1]`, and so on. None of the functions which operate on `filter_fir_s16_t` structs in this library will modify the contents of the buffer to which `coef` points. This buffer must be at least `num_taps` elements long, and must begin at a word-aligned address.

`state` is a pointer to a buffer (supplied by the user at initialization) containing the state data — a history of the `num_taps` most recent input samples. `state` must begin at a word-aligned address.

Coefficient Scaling

Filter Conversion

This library includes a python script which converts existing floating-point FIR filter coefficients into a suitable representation and generates code for easily initializing and executing the filter. See *Note: Digital Filter Conversion* for more.

Todo:

Usage Example

See also:

filter_fir_s16_init, *filter_fir_s16_add_sample*, *filter_fir_s16*

struct **filter_biquad_s32_t**

#include <filter.h> A biquad filter block.

Contains the coeffient and state information for a cascade of up to 8 biquad filter sections.

To process a new input sample, *filter_biquad_s32()* can be used with a pointer to one of these structs.

For longer cascades, an array of `filter_biquad_s32_t` structs can be used with *filter_biquads_s32()*.

Filter Conversion

This library includes a python script which converts existing floating-point cascaed biquad filter coefficients into a suitable representation and generates code for easily initializing and executing the filter. See *Note: Digital Filter Conversion* for more.

4.6 Scalar API

4.6.1 Scalar API Quick Reference

- *Scalar Type Conversion*
- *Fixed-Point Scalar Ops*
- *IEEE 754 Float Scalar Ops*
- *Non-standard Float Scalar Ops*
- *Non-standard Complex Float Scalar Ops*

4.6.1.1 Scalar Type Conversion

Table 4.11: Scalar Type Conversion

Function	Type In	Type Out
`f32_unpack()`	`float`	`int32_t`, *exponent_t*
`f32_unpack_s16()`	`float`	`int16_t`, *exponent_t*
`f32_to_float_s32()`	`float`	*float_s32_t*
`f64_to_float_s32()`	`double`	*float_s32_t*
`float_s32_to_float_s64()`	*float_s32_t*	*float_s64_t*
`float_s32_to_float()`	*float_s32_t*	`float`
`float_s32_to_double()`	*float_s32_t*	`double`
`s16_to_s32()`	`int16_t`, *exponent_t*	`int32_t`, *exponent_t*
`s32_to_s16()`	`int32_t`, *exponent_t*	`int16_t`, *exponent_t*
`s64_to_s32()`	`int64_t`, *exponent_t*	`int32_t`, *exponent_t*
`s32_to_f32()`	`int32_t`, *exponent_t*	`float`
`radians_to_sbrads()`	*radian_q24_t*	*sbrad_t*
`s32_to_chunk_s32()`	`int32_t`	`int32_t[8]`
`float_s64_to_float_s32()`	*float_s64_t*	*float_s32_t*

4.6.1.2 Fixed-Point Scalar Ops

Table 4.12: Fixed-Point Scalar Ops

Function	Input Depth	Fractional Bits	Brief
`s16_inverse()`	16	0	x^{-1}
`s32_inverse()`	32	0	x^{-1}
`sbrad_sin()`	32	31	$\sin(x)$

Table 4.12 – continued from previous page

Function	Input Depth	Fractional Bits	Brief
sbrad_tan()	32	31	$\tan(x)$
q24_sin()	32	24	$\sin(x)$
q24_cos()	32	24	$\cos(x)$
q24_tan()	32	24	$\tan(x)$
q30_exp_small()	32	30	$\exp(x)$
q24_logistic()	32	24	$\frac{1}{1+e^{-x}}$
q24_logistic_fast()	32	24	$\frac{1}{1+e^{-x}}$
q30_powers()	32	30	$(0, x, x^2, x^3, \dots)$
u32_ceil_log2()	32	0	$\lceil \log_2(x) \rceil$

4.6.1.3 IEEE 754 Float Ops

Table 4.13: IEEE 754 Float Ops

Function	Brief
f32_sin()	$sin(x)$
f32_cos()	$cos(x)$
f32_log2()	$log_2(x)$
f32_power_series()	Evaluate Power Series
f32_normA()	Normalized Form A

4.6.1.4 Non-standard Scalar Float Ops

Table 4.14: Non-standard Scalar Float Ops

Function	Brief
float_s32_mul()	$x \times y$
float_s32_add()	$x + y$
float_s32_sub()	$x - y$
float_s32_div()	$\frac{x}{y}$
float_s32_abs()	$|x|$
float_s32_gt()	$x > y$
float_s32_gte()	$x \geq y$
float_s32_ema()	$\alpha x + (1 - \alpha)y$
float_s32_sqrt()	$\sqrt{x}$
float_s32_exp()	$exp(x)$
s16_mul()	$x \times y$
s32_sqrt()	$\sqrt{x}$
s32_mul()	$x \times y$
s32_odd_powers()	$x, x^3, x^5, x^7, \dots$

4.6.1.5 Non-standard Complex Scalar Float Ops

Table 4.15: Non-standard Complex Scalar Float Ops

Function	Brief
float_complex_s16_mul()	$x \times y$
float_complex_s16_add()	$x + y$
float_complex_s16_sub()	$x - y$
float_complex_s32_mul()	$x \times y$
float_complex_s32_add()	$x + y$
float_complex_s32_sub()	$x - y$

4.6.2 16-bit Scalar API

group `scalar_s16_api`

Functions

int32_t **s16_to_s32**(*exponent_t* *a_exp, const int16_t b, const *exponent_t* b_exp, const unsigned remove_hr)

Convert a 16-bit floating-point scalar to a 32-bit floating-point scalar.

Converts a 16-bit floating-point scalar, represented by the 16-bit mantissa `b` and exponent `b_exp`, into a 32-bit floating-point scalar, represented by the 32-bit returned mantissa and output exponent `a_exp`.

`remove_hr`, if nonzero, indicates that the output mantissa should have no headroom. Otherwise, the output mantissa will be the same as the input mantissa.

Parameters

- `a_exp` – **[out]** Output exponent
- `b` – **[in]** 16-bit input mantissa
- `b_exp` – **[in]** Input exponent
- `remove_hr` – **[in]** Whether to remove headroom in output

Returns
32-bit output mantissa

int16_t **s16_inverse**(*exponent_t* *a_exp, const int16_t b)

Compute the inverse of a 16-bit integer.

`b` represents the integer b. `a` and `a_exp` together represent the result $a \cdot 2^{a\_exp}$.

Operation Performed

$$a \cdot 2^{a\_exp} \leftarrow \frac{1}{b}$$

Parameters

- `a_exp` – **[out]** Output exponent a_exp
- `b` – **[in]** Input integer b

Returns
Output mantissa a

int16_t **s16_mul**(*exponent_t* *a_exp, const int16_t b, const int16_t c, const *exponent_t* b_exp, const *exponent_t* c_exp)

Compute the product of two 16-bit floating-point scalars.

a and a_exp together represent the result $a \cdot 2^{a\_exp}$.

b and b_exp together represent the result $b \cdot 2^{b\_exp}$.

c and c_exp together represent the result $c \cdot 2^{c\_exp}$.

Operation Performed

$$a \cdot 2^{a\_exp} \leftarrow \left(b \cdot 2^{b\_exp}\right) \cdot \left(c \cdot 2^{c\_exp}\right)$$

Parameters

- **a_exp** − **[out]** Output exponent a_exp
- **b** − **[in]** First input mantissa b
- **c** − **[in]** Second input mantissa c
- **b_exp** − **[in]** First input exponent b_exp
- **c_exp** − **[in]** Second input exponent c_exp

Returns
Output mantissa a

4.6.3   32-bit Scalar API

group **scalar_s32_api**

Defines

S32_SQRT_MAX_DEPTH
Maximum bit-depth to calculate with *s32_sqrt()*.

Functions

float **s32_to_f32**(const int32_t mantissa, const *exponent_t* exp)
Pack a floating point value into an IEEE 754 single-precision float.

The value returned is the nearest representable approximation to $m \cdot 2^p$ where m is mantissa and p is exp.

Example

```
// Pack -12345678 * 2^{-13} into a float
int32_t mant = -12345678;
exponent_t exp = -13;
float val = s32_to_f32(mant, exp);

printf("%e <-- %ld * 2^(%d)\n", val, mant, exp);
```

Note: This operation may result in a loss of precision.

Parameters

- **mantissa** – **[in]** Mantissa of value to be packed

- **exp** – **[in]** Exponent of value to be packed

Returns

    `float` representation of input value

int16_t **s32_to_s16**(*exponent_t* *a_exp, const int32_t b, const *exponent_t* b_exp)

    Convert a 32-bit floating-point scalar to a 16-bit floating-point scalar.

    Converts a 32-bit floating-point scalar, represented by the 32-bit mantissa `b` and exponent `b_exp`, into a 16-bit floating-point scalar, represented by the 16-bit returned mantissa and output exponent `a_exp`.

Parameters

- **a_exp** – **[out]** Output exponent

- **b** – **[in]** 32-bit input mantissa

- **b_exp** – **[in]** Input exponent

Returns

    16-bit output mantissa

int32_t **s32_sqrt**(*exponent_t* *a_exp, const int32_t b, const *exponent_t* b_exp, const unsigned depth)

    Compute the square root of a 32-bit floating-point scalar.

    `b` and `b_exp` together represent the input $b \cdot 2^{b\_exp}$. Likewise, `a` and `a_exp` together represent the result $a \cdot 2^{a\_exp}$.

    `depth` indicates the number of MSb's which will be calculated. Smaller values here will execute more quickly at the cost of reduced precision. The maximum valid value for `depth` is *S32_SQRT_MAX_DEPTH*.

Operation Performed

$$a \cdot 2^{a\_exp} \leftarrow \sqrt{(b \cdot 2^{b\_exp})}$$

Parameters

- **a_exp** – **[out]** Output exponent a_exp

- **b** – **[in]** Input mantissa b

- **b_exp** – **[in]** Input exponent b_exp

- **depth** – **[in]** Number of most significant bits to calculate

Returns

    Output mantissa a

int32_t **s32_inverse**(*exponent_t* *a_exp, const int32_t b)

Compute the inverse of a 32-bit integer.

b represents the integer b. a and a_exp together represent the result $a \cdot 2^{a\_exp}$.

Operation Performed

$$a \cdot 2^{a\_exp} \leftarrow \frac{1}{b}$$

If b is the mantissa of a fixed- or floating-point value with an implicit or explicit exponent b_exp, then

Fixed- or Floating-point

$$\frac{1}{b \cdot 2^{b\_exp}} = \frac{1}{b} \cdot 2^{-b\_exp}$$
$$= a \cdot 2^{a\_exp} \cdot 2^{-b\_exp}$$
$$= a \cdot 2^{a\_exp - b\_exp}$$

and so b_exp should be subtracted from the output exponent a_exp.

Parameters

- **a_exp** − **[out]** Output exponent a_exp
- **b** − **[in]** Input integer b

Returns

Output mantissa a

int32_t **s32_mul**(*exponent_t* *a_exp, const int32_t b, const int32_t c, const *exponent_t* b_exp, const *exponent_t* c_exp)

Compute the product of two 32-bit floating-point scalars.

a and a_exp together represent the result $a \cdot 2^{a\_exp}$.

b and b_exp together represent the result $b \cdot 2^{b\_exp}$.

c and c_exp together represent the result $c \cdot 2^{c\_exp}$.

Operation Performed

$$a \cdot 2^{a\_exp} \leftarrow \left(b \cdot 2^{b\_exp}\right) \cdot \left(c \cdot 2^{c\_exp}\right)$$

Parameters

- **a_exp** − **[out]** Output exponent a_exp
- **b** − **[in]** First input mantissa b
- **c** − **[in]** Second input mantissa c
- **b_exp** − **[in]** First input exponent b_exp
- **c_exp** − **[in]** Second input exponent c_exp

Returns

Output mantissa a

sbrad_t **radians_to_sbrads**(const *radian_q24_t* theta)

Convert angle from radians to a modified binary representation.

Some trig functions, such as *sbrad_sin()*, rather than taking an angle specified in radians (e.g. *radian_q24_t*), require their argument to be a modified representation of the angle, as an *sbrad_t*. The modified binary representation takes into account various properies of the $sin(\theta)$ function to simplify certain operations.

For any angle θ there is a unique angle α where $-1 \leq \alpha \leq 1$ and $sin(\frac{\pi}{2}\alpha) = sin(\theta)$. This function essentially just maps the input angle θ onto the corresponding angle α in that region and returns the result in a Q1.31 format.

In this library, the unit of the resulting angle α is referred to as an 'sbrad'. 'brad' because α is a kind of binary angular measurement, and 's' because the symmetries of $sin(\theta)$ are what's being accounted for.

Parameters

- **theta** − **[in]** Input angle θ, in radians (Q8.24)

Returns

Output angle α, in sbrads

q2_30 **sbrad_sin**(const *sbrad_t* theta)

Compute the sine of the specified angle.

This function computes $sin(\frac{\pi}{2}\theta)$, returning the result in Q2.30 format.

The input angle θ must be expressed in sbrads (*sbrad_t*), and must represent a value between ± 0.5 (inclusive) (as a Q1.31).

Operation Performed

$$sin(\frac{\pi}{2}\theta)$$

Parameters

- **theta** − **[in]** Input angle θ, in sbrads (see *radians_to_sbrads*)

Returns

Sine of the specified angle in Q2.30 format.

q2_30 **sbrad_tan**(const *sbrad_t* theta)

Compute the tangent of the specified angle.

This function computes $tan(\frac{\pi}{2}\theta)$, returning the result in Q2.30 format.

The input angle θ must be expressed in sbrads (*sbrad_t*), and must represent a value between ± 0.25 (inclusive) (as a Q1.31).

Operation Performed

$$tan(\frac{\pi}{2}\theta)$$

Parameters

- **theta** − **[in]** Input angle θ, in sbrads (see *radians_to_sbrads*)

Returns

Tangent of the specified angle in Q2.30 format.

q2_30 **q24_sin**(const *radian_q24_t* theta)

Compute the sine of the specified angle.

This function computes $sin(\theta)$, returning the result in Q2.30 format.

Operation Performed

$$sin(\theta)$$

Parameters

- **theta** – **[in]** Input angle θ, in radians (Q8.24)

Returns
$sin(\theta)$ as a Q2.30

q2_30 **q24_cos**(const *radian_q24_t* theta)

Compute the cosine of the specified angle.

This function computes $cos(\theta)$, returning the result in Q2.30 format.

Operation Performed

$$cos(\theta)$$

Parameters

- **theta** – **[in]** Input angle θ, in radians (Q8.24)

Returns
$cos(\theta)$ as a Q2.30

float_s32_t **q24_tan**(const *radian_q24_t* theta)

Compute the tangent of the specified angle.

This function computes $tan(\theta)$. The result is returned as a *float_s32_t* containing a mantissa and exponent.

The value of $tan(\theta)$ is considered undefined where $theta = \frac{\pi}{2} + k\pi$ for any integer k. An exception will be raised if θ meets this condition.

Operation Performed

$$tan(\theta)$$

Parameters

- **theta** – **[in]** Input angle θ, in radians (Q8.24)

Throws ET_ARITHMETIC
Raised if $tan(\theta)$ is undefined.

Returns
$tan(\theta)$ as a *float_s32_t*

q2_30 **q30_exp_small**(const *q2_30* x)

Compute e^x for Q2.30 value near 0.

This function computes e^x where x is a fixed-point value with 30 fractional bits.

This function implements e^x using a truncated power series, and is only intended to be used for inputs in the range $-0.5 \leq x \leq 0.5$.

The output is also in the Q2.30 format.

For the range $-0.5 \leq x \leq 0.5$, the maximum observed error (compared to `exp(double)` from `math.h`) was `2` (which corresponds to 2^{-29}).

For the range $-1.0 \leq x \leq 1.0$, the corresponding maximum observed error was `324`, or approximately 2^{-21}.

To compute e^x for x outside of $[-0.5, 0.5]$, use *float_s32_exp()*.

Operation Performed

$$y \leftarrow e^x$$

Parameters

- **x** − **[in]** Input value x

Returns

y

q8_24 **q24_logistic**(const *q8_24* x)

Evaluate the logistic function at the specified point.

This function computes the value of the logistic function $y = \frac{1}{1+e^{-x}}$. This is a sigmoidal curve bounded below by $y = 0$ and above by $y = 1$.

The input x and output y are both Q8.24 fixed-point values.

If speed is greatly preferred to precision, *q24_logistic_fast()* can be used instead.

Operation Performed

$$y \leftarrow \frac{1}{1 + e^{-x}}$$

Parameters

- **x** − **[in]** Input value x

Returns

y

q8_24 **q24_logistic_fast**(const *q8_24* x)

Evaluate the logistic function at the specified point.

This function computes the value of the logistic function $y = \frac{1}{1+e^{-x}}$. This is a sigmoidal curve bounded below by $y = 0$ and above by $y = 1$.

The input x and output y are both Q8.24 fixed-point values.

This implementation trades off precision for speed, approximating results in a piece-wise linear manner. If a precise result is desired, *q24_logistic()* should be used instead.

Operation Performed

$$y \leftarrow \frac{1}{1 + e^{-x}}$$

Parameters

- **x** − **[in]** Input value x

Returns

y

void **s32_to_chunk_s32**(int32_t a[VPU_INT32_EPV], int32_t b)

Broadcast an integer to a vector chunk.

This function broadcasts the input b to the 8 elements of $\bar{a}$.

Operation Performed

$$a_k \leftarrow b$$

Parameters

- **a** − **[out]** Output chunk $\bar{a}$
- **b** − **[in]** Input value b

Throws ET_LOAD_STORE

Raised if a is not double word-aligned (See *Note: Vector Alignment*)

void **q30_powers**(*q2_30* a[], const *q2_30* b, const unsigned N)

Get the first N powers of b.

This function computes the first N powers (starting with 0) of the Q2.30 input b. The results are output as $\bar{a}$, also in Q2.30 format.

Operation Performed

$$a_0 \leftarrow 2^{30} = \text{Q30(1.0)}$$
$$a_k \leftarrow round\left(\frac{a_{k-1} \cdot b}{2^{30}}\right)$$
$$\text{for } k \in 0..N-1$$

Parameters

- **a** − **[out]** Output $\bar{a}$
- **b** − **[in]** Input b
- **N** − **[in]** Number of elements of $\bar{a}$ to compute

void **s32_odd_powers**(int32_t a[], const int32_t b, const unsigned count, const *right_shift_t* shr)

Fill vector with odd powers of b.

This function populates the elements of output vector $\bar{a}$ with the odd powers of input b. The first count odd powers of b are output. The highest power output will be $2 \cdot \text{count} - 1$.

The 64-bit product of each multiplication is right-shifted by shr bits and truncated to the 32 least significant bits. If b is a fixed-point value with shr fractional bits, then each a_k will have the same Q-format as input b. shr must be non-negative.

This function neither rounds nor saturates results. It is up to the user to ensure overflows are avoided.

Typical use-case is computing a power series of a function with odd symmetry.

Operation Performed

$$b_{sqr} = \frac{b^2}{2^{\mathsf{shr}}}$$

$$a_0 \leftarrow b$$

$$a_k \leftarrow \frac{a_{k-1}, b_{sqr}}{\mathsf{shr}}$$

$$\text{for } k \in 1, 2, 3, ..., \mathsf{count} - 1$$

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** Input b
- **count** – **[in]** Number of elements to output.
- **shr** – **[in]** Number of bits to right-shift 64-bit products.

4.6.4 Scalar IEEE 754 Float API

group `scalar_f32_api`

Functions

void **f32_unpack**(int32_t *mantissa, *exponent_t* *exp, const float input)
    Unpack an IEEE 754 single-precision float into a 32-bit mantissa and exponent.

Example

```
// Unpack 1.52345246 * 10^(-5)
float val = 1.52345246e-5;
int32_t mant;
exponent_t exp;
f32_unpack(&mant, &exp, val);

printf("%ld * 2^(%d) <-- %e\n", mant, exp, val);
```

Parameters

- **mantissa** – **[out]** Unpacked output mantissa
- **exp** – **[out]** Unpacked output exponent
- **input** – **[in]** Float value to be unpacked

void **f32_unpack_s16**(int16_t *mantissa, *exponent_t* *exp, const float input)
    Unpack an IEEE 754 single-precision float into a 16-bit mantissa and exponent.

Example

```
// Unpack 1.52345246 * 10^(-5)
float val = 1.52345246e-5;
int16_t mant;
exponent_t exp;
f32_unpack_s16(&mant, &exp, val);

printf("%ld * 2^(%d) <-- %e\n", mant, exp, val);
```

Note: This operation may result in a loss of precision.

Parameters

- **mantissa** – **[out]** Unpacked output mantissa
- **exp** – **[out]** Unpacked output exponent
- **input** – **[in]** Float value to be unpacked

float_s32_t **f32_to_float_s32**(const float x)

Convert an IEEE754 `float` to a *float_s32_t*.

Parameters

- **x** – **[in]** Input value

Throws ET_ARITHMETIC
Raised if `x` is infinite or NaN

Returns
`float_s32_t` representation of `x`

float_s32_t **f64_to_float_s32**(const double x)

Convert an IEEE754 `double` to a *float_s32_t*.

Note: This operation may result in precision loss.

Parameters

- **x** – **[in]** Input value

Throws ET_ARITHMETIC
Raised if `x` is infinite or NaN

Returns
`float_s32_t` representation of `x`

float **f32_sin**(const float theta)

Get the sine of a specified angle.

Computes $sin(\theta)$ using the power series expansion of $sin()$ truncated to 8 terms.

This implementation is meant to make optimal use of the XS3 floating-point unit.

Parameters

- **theta** – **[in]** Angle θ to compute the sine of (in radians)

Throws ET_ARITHMETIC
Raised if θ is infinite or NaN

Returns
Sine of the angle θ

float **f32_cos**(const float theta)

> Get the cosine of a specified angle.
>
> Computes $cos(\theta) = sin(\theta + \frac{\pi}{2})$ using the power series expansion of $sin()$ truncated to 8 terms.
>
> This implementation is meant to make optimal use of the XS3 floating-point unit.
>
> ### Parameters
>
> > - **theta** – **[in]** Angle θ to compute the cosine of (in radians)
>
> ### Throws ET_ARITHMETIC
> > Raised if θ is infinite or NaN
>
> ### Returns
> > Cosine of the angle θ

float **f32_log2**(const float x)

> Get the base-2 logarithm of the specified value.
>
> This function computes $log_2(x)$ using the power series expansion of $log_2()$ truncated to 11 terms.
>
> ### Parameters
>
> > - **x** – **[in]** Input value x to get the logarithm of.
>
> ### Throws ET_ARITHMETIC
> > Raised if x is infinite or NaN
>
> ### Returns
> > $log_2(x)$

float **f32_power_series**(const float x, const float b[], const unsigned N)

> Compute power series summation using specified coefficients.
>
> This function is used to compute the sum of terms in a power series, truncated to N terms, starting with the x^0 term.
>
> b is an N-element vector of coefficients $\bar{b}$ which are multiplied by the corresponding powers of x.
>
> N is the length of $\bar{b}$ and number of terms to sum together.
>
> ### Operation Performed
>
> $$a \leftarrow \sum_{k=0}^{N-1} \left(x^k, b_k\right)$$
>
> ### Parameters
>
> > - **x** – **[in]** Input value x.
> > - **b** – **[in]** Vector of coefficients $\bar{b}$.
> > - **N** – **[in]** Number of power series terms to sum.
>
> ### Throws ET_ARITHMETIC
> > Raised if x or any element of $\bar{b}$ is infinite or NaN.
>
> ### Returns
> > a, the sum of the first N power series terms.

float **f32_normA**(*exponent_t* *p, const float x)

> Get a representation of the input x in normalized form A.
>
> This function is used internally to transform a `float` value into a representation required for certain purposes.

In particular, this function behaves much like `frexpf()`, where it is guaranteed that the returned value a is either 0 or that $0.5 \leq |a| < 1.0$, and the output exponent p is such that $x = a \cdot 2^p$.

In anticipation that future work may require alternative "normalized" representations, this form is being defined here as form A.

Parameters

- **p** – **[in]** Output exponent p
- **x** – **[in]** Input value x

Throws ET_ARITHMETIC

Raised if x or any element of $\bar{b}$ is infinite or NaN.

Returns

a in normalized form A.

4.6.5 32-bit Scalar Float API

group `float_s32_api`

Functions

float_s64_t **float_s32_to_float_s64**(const *float_s32_t* x)

Convert a *float_s32_t* to a *float_s64_t*.

Parameters

- **x** – **[in]** Input value

Returns

`float_s64_t` representation of x

float **float_s32_to_float**(const *float_s32_t* x)

Convert a *float_s32_t* to an IEEE754 `float`.

Parameters

- **x** – **[in]** Input value

Returns

`float` representation of x

double **float_s32_to_double**(const *float_s32_t* x)

Convert a *float_s32_t* to an IEEE754 `double`.

Parameters

- **x** – **[in]** Input value

Returns

`double` representation of x

float_s32_t **float_s32_mul**(const *float_s32_t* x, const *float_s32_t* y)

Multiply two *float_s32_t* together.

The inputs x and y are multiplied together for a result a, which is returned.

Operation Performed

$$a \leftarrow x \cdot y$$

Parameters

- **x** – **[in]** Input operand x
- **y** – **[in]** Input operand y

Returns
>The product of x and y

float_s32_t **float_s32_add**(const *float_s32_t* x, const *float_s32_t* y)
>Add two *float_s32_t* together.

>The inputs x and y are added together for a result a, which is returned.

>**Operation Performed**

$$a \leftarrow x + y$$

>**Parameters**

>- **x** – **[in]** Input operand x
>- **y** – **[in]** Input operand y

>**Returns**
>>The sum of x and y

float_s32_t **float_s32_sub**(const *float_s32_t* x, const *float_s32_t* y)
>Subtract one *float_s32_t* from another.

>The input y is subtracted from the input x for a result a, which is returned.

>**Operation Performed**

$$a \leftarrow x - y$$

>**Parameters**

>- **x** – **[in]** Input operand x
>- **y** – **[in]** Input operand y

>**Returns**
>>The difference of x and y

float_s32_t **float_s32_div**(const *float_s32_t* x, const *float_s32_t* y)
>Divide one *float_s32_t* from another.

>The input x is divided by the input y for a result a, which is returned.

>**Operation Performed**

$$a \leftarrow \frac{x}{y}$$

>**Parameters**

>- **x** – **[in]** Input operand x
>- **y** – **[in]** Input operand y

Throws ET_ARITHMETIC
    if Y is 0

Returns
    The result of x/y

float_s32_t **float_s32_abs**(const *float_s32_t* x)

    Get the absolute value of a *float_s32_t*.

    a, the absolute value of x is returned.

Operation Performed

$$a \leftarrow |x|$$

Parameters

- **x** – **[in]** Input operand x

Returns
    The absolute value of x

unsigned **float_s32_gt**(const *float_s32_t* x, const *float_s32_t* y)

    Determine whether one *float_s32_t* is greater than another.

    The inputs x and y are compared. The result a is true iff x is greater than y and false otherwise. a is returned.

Operation Performed

$$a \leftarrow \begin{cases} 1 & x > y \\ 0 & otherwise \end{cases}$$

Parameters

- **x** – **[in]** Input operand x
- **y** – **[in]** Input operand y

Returns
    1 iff $x > y$; 0 otherwise

unsigned **float_s32_gte**(const *float_s32_t* x, const *float_s32_t* y)

    Determine whether one *float_s32_t* is greater or equal to another.

    The inputs x and y are compared. The result a is true iff x is greater than or equal to y and false otherwise. a is returned.

Operation Performed

$$a \leftarrow \begin{cases} 1 & x \geq y \\ 0 & otherwise \end{cases}$$

Parameters

- **x** – **[in]** Input operand x
- **y** – **[in]** Input operand y

Returns

1 iff $x \geq y$; 0 otherwise

float_s32_t **float_s32_ema**(const *float_s32_t* x, const *float_s32_t* y, const *uq2_30* coef)

Update an exponential moving average.

This function updates an exponential moving average by applying a single new sample. x is taken as the previous EMA state, with y as the new sample. The EMA coefficient α is applied to the term including x.

`coef` is a fixed-point value in a UQ2.30 format (i.e. has an implied exponent of -30), and should be in the range $0 \leq \alpha \leq 1$.

Operation Performed

$$a \leftarrow \alpha \cdot x + (1 - \alpha) \cdot y$$

Parameters

- **x** − **[in]** Input operand x
- **y** − **[in]** Input operand y
- **coef** − **[in]** EMA coefficient α encoded in UQ2.30 format

Returns

The new EMA state

float_s32_t **float_s32_sqrt**(const *float_s32_t* x)

Get the square root of a *float_s32_t*.

This function computes the square root of x. The result, a is returned.

The precision with which a is computed is configurable via the *XMATH_BFP_SQRT_DEPTH_S32* configuration parameter. It indicates the number of most significant bits to be calculated.

Operation Performed

$$a \leftarrow \sqrt{x}$$

See also:

XMATH_BFP_SQRT_DEPTH_S32

Warning: x must be non-negative to get a correct result.

Parameters

- **x** − **[in]** Input operand x

Returns

The square root of x

float_s32_t **float_s32_exp**(const *float_s32_t* x)

> Compute e^x.
>
> This function computes e^x for real input x.
>
> If x is known to be in the interval $[-0.5, 0.5]$, *q30_exp_small()* (which is used internally by this function) may be used instead for a speed boost.
>
> **Operation Performed**
>
> $$y \leftarrow e^x$$
>
> > **Parameters**
> >
> > - **x** − **[in]** Input x
> >
> > **Returns**
> > > y

float_s32_t **float_s64_to_float_s32**(const *float_s64_t* x)

> Convert a *float_s64_t* to a *float_s32_t*.
>
> ---
> **Note:** This operation may result in precision loss.
>
> ---
>
> > **Parameters**
> >
> > - **x** − **[in]** Input value
> >
> > **Returns**
> > > *float_s32_t* representation of x

4.6.6 16-bit Complex Scalar Floating-Point API

group **float_complex_s16_api**

Functions

float_complex_s16_t **float_complex_s16_mul**(const *float_complex_s16_t* x, const *float_complex_s16_t* y)

> Multiply two *float_complex_s16_t* together.
>
> The inputs x and y are multiplied together (using complex multiplication) for a result a, which is returned.
>
> **Operation Performed**
>
> $$a \leftarrow x \cdot y$$
>
> > **Parameters**
> >
> > - **x** − **[in]** Input operand x
> > - **y** − **[in]** Input operand y

Returns
    a, the complex product of x and y

float_complex_s16_t **float_complex_s16_add**(const *float_complex_s16_t* x, const *float_complex_s16_t* y)

Add two *float_complex_s16_t* together.

The inputs x and y are added together for a result a, which is returned.

Operation Performed

$$a \leftarrow x + y$$

Parameters

- **x** – **[in]** Input operand x
- **y** – **[in]** Input operand y

Returns
    a, the sum of x and y

float_complex_s16_t **float_complex_s16_sub**(const *float_complex_s16_t* x, const *float_complex_s16_t* y)

Subtract one *float_complex_s16_t* from another.

The input y is subtracted from the input x for a result a, which is returned.

Operation Performed

$$a \leftarrow x - y$$

Parameters

- **x** – **[in]** Input operand x
- **y** – **[in]** Input operand y

Returns
    a, the difference of x and y

4.6.7   32-bit Complex Scalar Floating-Point API

group **float_complex_s32_api**

Functions

float_complex_s32_t **float_complex_s32_mul**(const *float_complex_s32_t* x, const *float_complex_s32_t* y)

Multiply two *float_complex_s32_t* together.

The inputs x and y are multiplied together (using complex multiplication) for a result a, which is returned.

Operation Performed

$$a \leftarrow x \cdot y$$

Parameters

- **x** − **[in]** Input operand x
- **y** − **[in]** Input operand y

Returns
  a, the complex product of x and y

float_complex_s32_t **float_complex_s32_add**(const *float_complex_s32_t* x, const *float_complex_s32_t* y)

Add two *float_complex_s32_t* together.

The inputs x and y are added together for a result a, which is returned.

Operation Performed

$$a \leftarrow x + y$$

Parameters

- **x** − **[in]** Input operand x
- **y** − **[in]** Input operand y

Returns
  a, the sum of x and y

float_complex_s32_t **float_complex_s32_sub**(const *float_complex_s32_t* x, const *float_complex_s32_t* y)

Subtract one *float_complex_s32_t* from another.

The input y is subtracted from the input x for a result a, which is returned.

Operation Performed

$$a \leftarrow x - y$$

Parameters

- **x** − **[in]** Input operand x
- **y** − **[in]** Input operand y

Returns
  a, the difference of x and y

4.6.8 Miscellaneous Scalar API

group `scalar_misc_api`

Functions

static inline unsigned **u32_ceil_log2**(unsigned N)

> Get the size of a 32-bit unsigned number.
>
> This function reports the size of the number as a, the number of bits required to store unsigned integer N. This is equivalent to $ceil\left(log_2\left(N\right)\right)$.
>
> N is the input N.

Operation Performed

$$a \leftarrow \begin{cases} 0 & N = 0 \\ \lceil log_2\left(N\right) \rceil & otherwise \end{cases}$$

> ##### Parameters
>
> - **N** − **[in]** Number to get the size of
>
> ##### Returns
>> Number of bits a required to store N

int32_t **s64_to_s32**(*exponent_t* *a_exp, const int64_t b, const *exponent_t* b_exp)

> Convert a 64-bit floating-point scalar to a 32-bit floating-point scalar.
>
> Converts a 64-bit floating-point scalar, represented by the 64-bit mantissa b and exponent b_exp, into a 32-bit floating-point scalar, represented by the 32-bit returned mantissa and output exponent a_exp.
>
> ##### Parameters
>
> - **a_exp** − **[out]** Output exponent
>
> - **b** − **[in]** 64-bit input mantissa
>
> - **b_exp** − **[in]** Input exponent
>
> ##### Returns
>> 32-bit output mantissa

4.7 Vector API

4.7.1 Vector API Quick Reference

The tables below list the functions of the vector API. The "EW" column indicates whether the operation acts element-wise.

The "Signature" column is intended as a hint which quickly conveys the kind of the conceptual inputs to and outputs from the operation. The signatures are only intended to convey how many (conceptual) inputs and outputs there are, and their dimensionality.

The functions themselves will typically take more arguments than these signatures indicate. For example, most functions take vector lengths as input, and many take shift values which are used to control growth of element bit-depth. Check the function's full documentation to get more detailed information.

The following symbols are used in the signatures:

Symbol	Description
$\mathbb{S}$	A scalar input or output value.
$\mathbb{V}$	A vector-valued input or output.
$\mathbb{M}$	A matrix-valued input or output.
$\varnothing$	Placeholder indicating no input or output.

For example, the operation signature $(\mathbb{V} \times \mathbb{V} \times \mathbb{S}) \to \mathbb{V}$ indicates the operation takes two vector inputs and a scalar input, and the output is a vector.

- *32-Bit Vector Ops*
- *16-Bit Vector Ops*
- *8-Bit Vector Ops*
- *Complex 32-Bit Vector Ops*
- *Complex 16-Bit Vector Ops*
- *Fixed-Point Vector Ops*
- *Floating-Point Vector Ops*
- *Other Vector Ops*
- *Vector Type Conversions*

32-bit Vector Ops

Function	EW	Signature
vect_s32_copy()		$\mathbb{V} \to \mathbb{V}$
vect_s32_abs()	x	$\mathbb{V} \to \mathbb{V}$
vect_s32_abs_sum()		$\mathbb{V} \to \mathbb{S}$
vect_s32_add()	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s32_add_scalar()	x	$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$
vect_s32_argmax()		$\mathbb{V} \to \mathbb{S}$
vect_s32_argmin()		$\mathbb{V} \to \mathbb{S}$
vect_s32_clip()	x	$(\mathbb{V} \times \mathbb{S} \times \mathbb{S}) \to \mathbb{V}$
vect_s32_dot()		$(\mathbb{V} \times \mathbb{V}) \to \mathbb{S}$
vect_s32_energy()		$\mathbb{V} \to \mathbb{S}$
vect_s32_headroom()		$\mathbb{V} \to \mathbb{S}$
vect_s32_inverse()	x	$\mathbb{V} \to \mathbb{V}$
vect_s32_max()		$\mathbb{V} \to \mathbb{S}$
vect_s32_max_elementwise()	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s32_min()		$\mathbb{V} \to \mathbb{S}$
vect_s32_min_elementwise()	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s32_mul()	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s32_macc()	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s32_nmacc()	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s32_rect()	x	$\mathbb{V} \to \mathbb{V}$
vect_s32_scale()	x	$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$
vect_s32_set()	x	$\mathbb{S} \to \mathbb{V}$
vect_s32_shl()	x	$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$
vect_s32_shr()	x	$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$

Table 4.17 – continued from previous page

32-bit Vector Ops Function	EW	Signature
vect_s32_sqrt()	x	$\mathbb{V} \to \mathbb{V}$
vect_s32_sub()	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s32_sum()		$\mathbb{V} \to \mathbb{S}$
vect_s32_zip()		$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s32_unzip()		$\mathbb{V} \to (\mathbb{V} \times \mathbb{V})$
vect_s32_convolve_valid()		$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s32_convolve_same()		$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s32_log_base()	x	$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$
vect_s32_log()	x	$\mathbb{V} \to \mathbb{V}$
vect_s32_log2()	x	$\mathbb{V} \to \mathbb{V}$
vect_s32_log10()	x	$\mathbb{V} \to \mathbb{V}$
chunk_s32_dot()		$(\mathbb{V} \times \mathbb{V}) \to \mathbb{S}$
chunk_s32_log()	x	$\mathbb{V} \to \mathbb{V}$

16-bit Vector Ops Function	EW	Signature
vect_s16_abs()	x	$\mathbb{V} \to \mathbb{V}$
vect_s16_abs_sum()		$\mathbb{V} \to \mathbb{S}$
vect_s16_add()	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s16_add_scalar()	x	$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$
vect_s16_argmax()		$\mathbb{V} \to \mathbb{S}$
vect_s16_argmin()		$\mathbb{V} \to \mathbb{S}$
vect_s16_clip()	x	$(\mathbb{V} \times \mathbb{S} \times \mathbb{S}) \to \mathbb{V}$
vect_s16_dot()		$(\mathbb{V} \times \mathbb{V}) \to \mathbb{S}$
vect_s16_energy()		$\mathbb{V} \to \mathbb{S}$
vect_s16_headroom()		$\mathbb{V} \to \mathbb{S}$
vect_s16_inverse()	x	$\mathbb{V} \to \mathbb{V}$
vect_s16_max()		$\mathbb{V} \to \mathbb{S}$
vect_s16_max_elementwise()	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s16_min()		$\mathbb{V} \to \mathbb{S}$
vect_s16_min_elementwise()	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s16_mul()	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s16_macc()	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s16_nmacc()	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s16_rect()	x	$\mathbb{V} \to \mathbb{V}$
vect_s16_scale()	x	$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$
vect_s16_set()	x	$\mathbb{S} \to \mathbb{V}$
vect_s16_shl()	x	$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$
vect_s16_shr()	x	$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$
vect_s16_sqrt()	x	$\mathbb{V} \to \mathbb{V}$
vect_s16_sub()	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_s16_sum()		$\mathbb{V} \to \mathbb{S}$
vect_s16_extract_high_byte()	x	$\mathbb{V} \to \mathbb{V}$
vect_s16_extract_low_byte()	x	$\mathbb{V} \to \mathbb{V}$

8-bit Vector Ops

Function	EW	Signature	Brief
vect_s8_is_negative()	x	$\mathbb{V} \rightarrow \mathbb{V}$	Identify negative elements

32-bit Complex Vector Ops

Function	EW	Signature
vect_complex_s32_add()	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s32_add_scalar()	x	$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$
vect_complex_s32_conj_macc()	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s32_conj_mul()	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s32_conj_nmacc()	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s32_conjugate()	x	$\mathbb{V} \rightarrow \mathbb{V}$
vect_complex_s32_headroom()		$\mathbb{V} \rightarrow \mathbb{S}$
vect_complex_s32_macc()	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s32_mag()	x	$\mathbb{V} \rightarrow \mathbb{V}$
vect_complex_s32_mul()	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s32_nmacc()	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s32_real_mul()	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s32_real_scale()	x	$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$
vect_complex_s32_scale()	x	$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$
vect_complex_s32_set()	x	$\mathbb{S} \rightarrow \mathbb{V}$
vect_complex_s32_shl()	x	$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$
vect_complex_s32_shr()	x	$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$
vect_complex_s32_squared_mag()	x	$\mathbb{V} \rightarrow \mathbb{V}$
vect_complex_s32_sub()	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s32_sum()		$\mathbb{V} \rightarrow \mathbb{S}$
vect_complex_s32_tail_reverse()		$\mathbb{V} \rightarrow \mathbb{V}$

16-bit Complex Vector Ops

Function	EW	Signature
vect_complex_s16_add()	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s16_add_scalar()	x	$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$
vect_complex_s16_conj_mul()	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s16_conj_macc()	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s16_conj_nmacc()	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s16_headroom()		$\mathbb{V} \rightarrow \mathbb{S}$
vect_complex_s16_macc()	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s16_mag()	x	$\mathbb{V} \rightarrow \mathbb{V}$
vect_complex_s16_mul()	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s16_nmacc()	x	$(\mathbb{V} \times \mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s16_real_mul()	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s16_real_scale()	x	$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$
vect_complex_s16_scale()	x	$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$
vect_complex_s16_set()	x	$\mathbb{S} \rightarrow \mathbb{V}$
vect_complex_s16_shl()	x	$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$
vect_complex_s16_shr()	x	$(\mathbb{V} \times \mathbb{S}) \rightarrow \mathbb{V}$
vect_complex_s16_squared_mag()	x	$\mathbb{V} \rightarrow \mathbb{V}$
vect_complex_s16_sub()	x	$(\mathbb{V} \times \mathbb{V}) \rightarrow \mathbb{V}$
vect_complex_s16_sum()		$\mathbb{V} \rightarrow \mathbb{S}$

Fixed-Point Vector Ops

Function	EW	Signature
vect_q30_power_series()	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
vect_q30_exp_small()	x	$\mathbb{V} \to \mathbb{V}$
chunk_q30_power_series()	x	$(\mathbb{V} \times \mathbb{V}) \to \mathbb{V}$
chunk_q30_exp_small()	x	$\mathbb{V} \to \mathbb{V}$

Floating-Point Vector Ops

Function	EW	Signature
vect_f32_max_exponent()		$\mathbb{V} \to \mathbb{S}$
vect_f32_dot()		$(\mathbb{V} \times \mathbb{V}) \to \mathbb{S}$
vect_f32_add()	x	$\mathbb{V} \times \mathbb{V} \to \mathbb{V}$
vect_float_s32_log_base()	x	$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$
vect_float_s32_log()	x	$\mathbb{V} \to \mathbb{V}$
vect_float_s32_log2()	x	$\mathbb{V} \to \mathbb{V}$
vect_float_s32_log10()	x	$\mathbb{V} \to \mathbb{V}$
chunk_float_s32_log()	x	$\mathbb{V} \to \mathbb{V}$
vect_complex_f32_add()	x	$\mathbb{V} \times \mathbb{V} \to \mathbb{V}$
vect_complex_f32_mul()	x	$\mathbb{V} \times \mathbb{V} \to \mathbb{V}$
vect_complex_f32_conj_mul()	x	$\mathbb{V} \times \mathbb{V} \to \mathbb{V}$
vect_complex_f32_macc()	x	$\mathbb{V} \times \mathbb{V} \times \mathbb{V} \to \mathbb{V}$
vect_complex_f32_conj_macc()	x	$\mathbb{V} \times \mathbb{V} \times \mathbb{V} \to \mathbb{V}$

Note that several of the functions below take vectors of the *split_acc_s32_t* type. This is a 32-bit vector type used for accumulating results of 8- or 16-bit operations in a manner optimized for the XS3 VPU.

Other Vector Ops

Function	EW	Signature
vect_split_acc_s32_shr()	x	$(\mathbb{V} \times \mathbb{S}) \to \mathbb{V}$
vect_s32_merge_accs()	x	$\mathbb{V} \to \mathbb{V}$
vect_s32_split_accs()	x	$\mathbb{V} \to \mathbb{V}$
chunk_s16_accumulate()	x	$\mathbb{V} \to \mathbb{V}$
mat_mul_s8_x_s8_yield_s32()		$(\mathbb{M} \times \mathbb{V}) \to \mathbb{V}$
mat_mul_s8_x_s16_yield_s32()		$(\mathbb{M} \times \mathbb{V}) \to \mathbb{V}$

Vector Type Conversion Ops

Function	Array Element Type	
	Input	Output
vect_s16_to_vect_s32()	int16_t	int32_t
vect_s32_to_vect_s16()	int32_t	int16_t
vect_s32_to_vect_f32()	int32_t	float
vect_f32_to_vect_s32()	float	int32_t
vect_complex_s16_to_vect_complex_s32()	*complex_s16_t*	*complex_s32_t*
vect_complex_s32_to_vect_complex_s16()	*complex_s32_t*	*complex_s16_t*

4.7.2   8-Bit Vector API

group `vect_s8_api`

Functions

void **vect_s8_is_negative**(int8_t a[], const int8_t b[], const unsigned length)

Determine whether each element of a signed 8-bit input vector are negative.

Each element a_k of 8-bit output vector $\bar{a}$ is set to 1 if the corresponding element b_k of 8-bit input vector $\bar{b}$ is negative, and is set to 0 otherwise.

a[] represents the 8-bit output vector $\bar{a}$, with the element a[k] representing a_k.

b[] represents the 8-bit input vector $\bar{b}$, with the element b[k] representing b_k.

length is the number of elements in a[] and b[].

Operation Performed

$$a_k \leftarrow \begin{cases} 1 & b_k < 0 \\ 0 & otherwise \end{cases}$$
$$\text{for } k \in 0 \dots (length - 1)$$

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **length** – **[in]** Number of elements in $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE

Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

void **mat_mul_s8_x_s8_yield_s32**(*split_acc_s32_t* accumulators[], const int8_t matrix[], const int8_t input_vect[], const unsigned M_rows, const unsigned N_cols)

Multiply-accumulate an 8-bit matrix by an 8-bit vector into 32-bit accumulators.

This function multiplies an 8-bit $M \times N$ matrix $\bar{W}$ by an 8-bit N-element column vector $\bar{v}$ and adds it to the 32-bit accumulator vector $\bar{a}$.

accumulators is the output vector $\bar{a}$ to which the product $\bar{W} \times \bar{v}$ is accumulated. Note that the accumulators are encoded in a format native to the xcore VPU. To initialize the accumulator vector to zeros, just zero the memory.

matrix is the matrix $\bar{W}$.

input_vect is the vector $\bar{v}$.

matrix and input_vect must both begin at a word-aligned offsets.

M_rows and N_rows are the dimensions M and N of matrix $\bar{W}$. M must be a multiple of 16, and N must be a multiple of 32.

The result of this multiplication is exact, so long as saturation does not occur.

Parameters

- **accumulators** – **[inout]** The accumulator vector $\bar{a}$
- **matrix** – **[in]** The weight matrix $\bar{W}$

- **input_vect** − **[in]** The input vector $\bar{v}$
- **M_rows** − **[in]** The number of rows M in matrix $\bar{W}$
- **N_cols** − **[in]** The number of columns N in matrix $\bar{W}$

Throws ET_LOAD_STORE
   Raised if `matrix` or `input_vect` is not word-aligned (See *Note: Vector Alignment*)

4.7.3 16-Bit Vector API

group **vect_s16_api**

Functions

headroom_t **vect_s16_abs**(int16_t a[], const int16_t b[], const unsigned length)
   Compute the element-wise absolute value of a 16-bit vector.

   `a[]` and `b[]` represent the 16-bit vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on `b[]`.

   `length` is the number of elements in each of the vectors.

   Operation Performed

   $$a_k \leftarrow sat_{32}(|b_k|)$$
   $$\text{for } k \in 0 \dots (length - 1)$$

   Block Floating-Point

   If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the output vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

   Parameters
   - **a** − **[out]** Output vector $\bar{a}$
   - **b** − **[in]** Input vector $\bar{b}$
   - **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

   Throws ET_LOAD_STORE
      Raised if `a` or `b` is not word-aligned (See *Note: Vector Alignment*)
   Returns
      Headroom of the output vector $\bar{a}$.

int32_t **vect_s16_abs_sum**(const int16_t b[], const unsigned length)
   Compute the sum of the absolute values of elements of a 16-bit vector.

   `b[]` represents the 16-bit vector $\bar{b}$. `b[]` must begin at a word-aligned address.

   `length` is the number of elements in $\bar{b}$.

Operation Performed

$$a \leftarrow \sum_{k=0}^{length-1} |b_k|$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the returned value a is the 32-bit mantissa of floating-point value $a \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Parameters

- **b** − **[in]** Input vector $\bar{b}$

- **length** − **[in]** Number of elements in $\bar{b}$

Throws ET_LOAD_STORE
Raised if b is not word-aligned (See *Note: Vector Alignment*)

Returns
The 32-bit sum a

headroom_t **vect_s16_add** (int16_t a[], const int16_t b[], const int16_t c[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Add one 16-bit BFP vector to another.

a[], b[] and c[] represent the 16-bit vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[] or c[].

length is the number of elements in each of the vectors.

b_shr and c_shr are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k = sat_{16}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k = sat_{16}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$a_k \leftarrow sat_{16}(b'_k + c'_k)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$.

In this case, b_shr and c_shr **must** be chosen so that $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$. Adding or subtracting mantissas only makes sense if they are associated with the same exponent.

The function *vect_s16_add_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

See also:

vect_s16_add_prepare

Parameters

- **a** – **[out]** Output vector $\bar{a}$

- **b** – **[in]** Input vector $\bar{b}$

- **c** – **[in]** Input vector $\bar{c}$

- **length** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$

- **b_shr** – **[in]** Right-shift appled to $\bar{b}$

- **c_shr** – **[in]** Right-shift appled to $\bar{c}$

Throws ET_LOAD_STORE
Raised if a, b or c is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$.

headroom_t **vect_s16_add_scalar**(int16_t a[], const int16_t b[], const int16_t c, const unsigned length, const *right_shift_t* b_shr)

Add a scalar to a 16-bit vector.

a[], b[] represent the 16-bit mantissa vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[].

c is the scalar c to be added to each element of $\bar{b}$.

length is the number of elements in each of the vectors.

b_shr is the signed arithmetic right-shifts applied to each element of $\bar{b}$.

Operation Performed

$$b'_k = sat_{16}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$a_k \leftarrow sat_{16}(b'_k + c)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If elements of $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, and c is the mantissa of floating-point value $c \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$.

In this case, b_shr and c_shr **must** be chosen so that $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$. Adding or subtracting mantissas only makes sense if they are associated with the same exponent.

The function *vect_s16_add_scalar_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Note that c_shr is an output of *vect_s16_add_scalar_prepare()*, but is not a parameter to this function. The c_shr produced by *vect_s16_add_scalar_prepare()* is to be applied by the user, and the result passed as input c.

See also:

vect_s16_add_scalar_prepare()

Parameters

- **a** – **[out]** Output vector $\bar{a}$

- **b** – **[in]** Input vector $\bar{b}$

- **c** – **[in]** Input scalar c

- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

- **b_shr** – **[in]** Right-shift appled to $\bar{b}$

Throws ET_LOAD_STORE
Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$.

unsigned **vect_s16_argmax**(const int16_t b[], const unsigned length)

Obtain the array index of the maximum element of a 16-bit vector.

b[] represents the 16-bit input vector $\bar{b}$. It must begin at a word-aligned address.

length is the number of elements in $\bar{b}$.

Operation Performed

$$a \leftarrow argmax_k\{b_k\}$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Parameters

- **b** – **[in]** Input vector $\bar{b}$

- **length** – **[in]** Number of elemetns in $\bar{b}$

Throws ET_LOAD_STORE
Raised if b is not word-aligned (See *Note: Vector Alignment*)

Returns
a, the index of the maximum element of vector $\bar{b}$. If there is a tie for the maximum value, the lowest tying index is returned.

unsigned **vect_s16_argmin**(const int16_t b[], const unsigned length)

Obtain the array index of the minimum element of a 16-bit vector.

b[] represents the 16-bit input vector $\bar{b}$. It must begin at a word-aligned address.

length is the number of elements in $\bar{b}$.

Operation Performed

$$a \leftarrow argmin_k\{b_k\}$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Parameters

- **b** – **[in]** Input vector $\bar{b}$

- **length** – **[in]** Number of elemetns in $\bar{b}$

Throws ET_LOAD_STORE
Raised if b is not word-aligned (See *Note: Vector Alignment*)

Returns
a, the index of the minimum element of vector $\bar{b}$. If there is a tie for the minimum value, the lowest tying index is returned.

headroom_t **vect_s16_clip**(int16_t a[], const int16_t b[], const unsigned length, const int16_t
lower_bound, const int16_t upper_bound, const *right_shift_t* b_shr)

Clamp the elements of a 16-bit vector to a specified range.

a[] and b[] represent the 16-bit vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[].

length is the number of elements in each of the vectors.

lower_bound and upper_bound are the lower and upper bounds of the clipping range respectively. These bounds are checked for each element of $\bar{b}$ only *after* b_shr is applied.

b_shr is the signed arithmetic right-shift applied to elements of $\bar{b}$ *before* being compared to the upper and lower bounds.

If $\bar{b}$ are the mantissas for a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the exponent a_exp of the output BFP vector $\bar{a} \cdot 2^{a\_exp}$ is given by $a\_exp = b\_exp + b\_shr$.

Operation Performed

$$b'_k \leftarrow sat_{16}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$

$$a_k \leftarrow \begin{cases} lower\_bound & b'_k \leq lower\_bound \\ upper\_bound & b'_k \geq upper\_bound \\ b'_k & otherwise \end{cases}$$

$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the output vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + b\_shr$.

Parameters

- **a** – **[out]** Output vector $\bar{a}$

- **b** – **[in]** Input vector $\bar{b}$

- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

- **lower_bound** – **[in]** Lower bound of clipping range

- **upper_bound** – **[in]** Upper bound of clipping range

- **b_shr** – **[in]** Arithmetic right-shift applied to elements of $\bar{b}$ prior to clipping

Throws ET_LOAD_STORE
Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of output vector $\bar{a}$

int64_t **vect_s16_dot**(const int16_t b[], const int16_t c[], const unsigned length)

Compute the inner product of two 16-bit vectors.

b[] and c[] represent the 32-bit vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address.

length is the number of elements in each of the vectors.

Operation Performed

$$a \leftarrow \sum_{k=0}^{length-1} (b_k \cdot c_k)$$

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the mantissas of the BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then result a is the mantissa of the result $a \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + c\_exp$.

If needed, the bit-depth of a can then be reduced to 16 or 32 bits to get a new result $a' \cdot 2^{a\_exp'}$ where $a' = a \cdot 2^{-a\_shr}$ and $a\_exp' = a\_exp + a\_shr$.

Notes

The sum a is accumulated simultaneously into 16 48-bit accumulators which are summed together at the final step. So long as `length` is less than roughly 2 million, no overflow or saturation of the resulting sum is possible.

Parameters

- **b** − **[in]** Input vector $\bar{b}$
- **c** − **[in]** Input vector $\bar{c}$
- **length** − **[in]** Number of elements in vectors $\bar{b}$ and $\bar{c}$

Throws ET_LOAD_STORE
Raised if b or c is not word-aligned (See *Note: Vector Alignment*)

Returns
a, the inner product of vectors $\bar{b}$ and $\bar{c}$.

int32_t **vect_s16_energy**(const int16_t b[], const unsigned length, const *right_shift_t* b_shr)

Calculate the energy (sum of squares of elements) of a 16-bit vector.

b[] represents the 16-bit vector $\bar{b}$. b[] must begin at a word-aligned address.

length is the number of elements in $\bar{b}$.

b_shr is the signed arithmetic right-shift applied to elements of $\bar{b}$. b_shr should be chosen to avoid the possibility of saturation. See the note below.

Operation Performed

$$b'_k \leftarrow sat_{16}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$

$$a \leftarrow \sum_{k=0}^{length-1} (b'_k)^2$$

Block Floating-Point

If $\bar{b}$ are the mantissas of the BFP vector $\bar{b} \cdot 2^{b\_exp}$, then floating-point result is $a \cdot 2^{a\_exp}$, where the 32-bit mantissa a is returned by this function, and $a\_exp = 2 \cdot (b\_exp + b\_shr)$.

Additional Details

If $\bar{b}$ has b_hr bits of headroom, then each product $(b'_k)^2$ can be a maximum of $2^{30-2\cdot(b\_hr+b\_shr)}$. So long as `length` is less than $1 + 2 \cdot (b\_hr + b\_shr)$, such errors should not be possible. Each increase of b_shr by 1 doubles the number of elements that can be summed without risk of overflow.

If the caller's mantissa vector is longer than that, the full result can be found by calling this function multiple times for partial results on sub-sequences of the input, and adding the results in user code.

In many situations the caller may have *a priori* knowledge that saturation is impossible (or very nearly so), in which case this guideline may be disregarded. However, such situations are application-specific and are well beyond the scope of this documentation, and as such are left to the user's discretion.

Parameters

- **b** − **[in]** Input vector $\bar{b}$

- **length** − **[in]** Number of elements in $\bar{b}$

- **b_shr** − **[in]** Right-shift appled to $\bar{b}$

Throws ET_LOAD_STORE
Raised if `b` is not word-aligned (See *Note: Vector Alignment*)

Returns
64-bit mantissa of vector $\bar{b}$'s energy

headroom_t **vect_s16_headroom**(const int16_t b[], const unsigned length)

Calculate the headroom of a 16-bit vector.

The headroom of an N-bit integer is the number of bits that the integer's value may be left-shifted without any information being lost. Equivalently, it is one less than the number of leading sign bits.

The headroom of an `int16_t` array is the minimum of the headroom of each of its `int16_t` elements.

This function efficiently traverses the elements of `b[]` to determine its headroom.

`b[]` represents the 16-bit vector $\bar{b}$. `b[]` must begin at a word-aligned address.

`length` is the number of elements in `b[]`.

Operation Performed

$$a \leftarrow min\{HR_{16}(x_0), HR_{16}(x_1), ..., HR_{16}(x_{length-1})\}$$

See also:

vect_s32_headroom, *vect_complex_s16_headroom*, *vect_complex_s32_headroom*

Parameters

- **b** − **[in]** Input vector $\bar{b}$

- **length** − **[in]** The number of elements in vector $\bar{b}$

Throws ET_LOAD_STORE
Raised if `b` is not word-aligned (See *Note: Vector Alignment*)

Returns
> Headroom of vector $\bar{b}$

void **vect_s16_inverse**(int16_t a[], const int16_t b[], const unsigned length, const unsigned scale)
> Compute the inverse of elements of a 16-bit vector.
>
> a[] and b[] represent the 16-bit mantissa vectors $\bar{a}$ and $\bar{b}$ respectively. This operation can be performed safely in-place on b[].
>
> length is the number of elements in each of the vectors.
>
> scale is a scaling parameter used to maximize the precision of the result.

> **Operation Performed**

$$a_k \leftarrow \lfloor \frac{2^{scale}}{b_k} \rfloor$$
$$\text{for } k \in 0 \ldots (length - 1)$$

> **Block Floating-Point**

> If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = scale - b\_exp$.
>
> The function *vect_s16_inverse_prepare()* can be used to obtain values for a_exp and $scale$.

> **See also:**
> *vect_s16_inverse_prepare*

> > **Parameters**
> > - **a** – **[out]** Output vector $\bar{a}$
> > - **b** – **[in]** Input vector $\bar{b}$
> > - **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
> > - **scale** – **[in]** Scale factor applied to dividend when computing inverse
> > **Returns**
> > > Headroom of output vector $\bar{a}$

int16_t **vect_s16_max**(const int16_t b[], const unsigned length)
> Find the maximum value in a 16-bit vector.
>
> b[] represents the 16-bit vector $\bar{b}$. It must begin at a word-aligned address.
>
> length is the number of elements in $\bar{b}$.

> **Operation Performed**

$$max\{x_0, x_1, \ldots, x_{length-1}\}$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the returned value a is the 16-bit mantissa of floating-point value $a \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Parameters

- **b** − **[in]** Input vector $\bar{b}$
- **length** − **[in]** Number of elements in $\bar{b}$

Throws ET_LOAD_STORE
Raised if b is not word-aligned (See *Note: Vector Alignment*)

Returns
Maximum value from $\bar{b}$

headroom_t **vect_s16_max_elementwise**(int16_t a[], const int16_t b[], const int16_t c[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Get the element-wise maximum of two 16-bit vectors.

a[], b[] and c[] represent the 16-bit mantissa vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[], but *not* on c[].

length is the number of elements in each of the vectors.

b_shr and c_shr are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k \leftarrow sat_{16}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k \leftarrow sat_{16}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$a_k \leftarrow max(b'_k, c'_k)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$.

The function *vect_2vec_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

> **Warning:** For correct operation, this function requires at least 1 bit of headroom in each mantissa vector *after* the shifts have been applied.

Parameters

- **a** − **[out]** Output vector $\bar{a}$
- **b** − **[in]** Input vector $\bar{b}$
- **c** − **[in]** Input vector $\bar{c}$
- **length** − **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **b_shr** − **[in]** Right-shift appled to $\bar{b}$

- **c_shr** − **[in]** Right-shift appled to $\bar{c}$

Throws ET_LOAD_STORE
> Raised if a, b or c is not word-aligned (See *Note: Vector Alignment*)

Returns
> Headroom of vector $\bar{a}$

int16_t **vect_s16_min**(const int16_t b[], const unsigned length)
> Find the minimum value in a 16-bit vector.
>
> b[] represents the 16-bit vector $\bar{b}$. It must begin at a word-aligned address.
>
> length is the number of elements in $\bar{b}$.

Operation Performed

$$max\{x_0, x_1, ..., x_{length-1}\}$$

Block Floating-Point

> If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the returned value a is the 16-bit mantissa of floating-point value $a \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Parameters

- **b** − **[in]** Input vector $\bar{b}$
- **length** − **[in]** Number of elements in $\bar{b}$

Throws ET_LOAD_STORE
> Raised if b is not word-aligned (See *Note: Vector Alignment*)

Returns
> Minimum value from $\bar{b}$

headroom_t **vect_s16_min_elementwise**(int16_t a[], const int16_t b[], const int16_t c[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)
> Get the element-wise minimum of two 16-bit vectors.
>
> a[], b[] and c[] represent the 16-bit mantissa vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[], but *not* on c[].
>
> length is the number of elements in each of the vectors.
>
> b_shr and c_shr are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k \leftarrow sat_{16}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k \leftarrow sat_{16}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$a_k \leftarrow min(b'_k, c'_k)$$
$$\text{for } k \in 0 \, ... \, (length - 1)$$

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$.

The function *vect_2vec_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

> **Warning:** For correct operation, this function requires at least 1 bit of headroom in each mantissa vector *after* the shifts have been applied.

Parameters

- **a** − **[out]** Output vector $\bar{a}$
- **b** − **[in]** Input vector $\bar{b}$
- **c** − **[in]** Input vector $\bar{c}$
- **length** − **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **b_shr** − **[in]** Right-shift appled to $\bar{b}$
- **c_shr** − **[in]** Right-shift appled to $\bar{c}$

Throws ET_LOAD_STORE

Raised if a, b or c is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of vector $\bar{a}$

headroom_t **vect_s16_macc**(int16_t acc[], const int16_t b[], const int16_t c[], const unsigned length, const *right_shift_t* acc_shr, const *right_shift_t* bc_sat)

Multiply one 16-bit vector element-wise by another, and add the result to an accumulator.

acc[] represents the 16-bit accumulator mantissa vector $\bar{a}$. Each a_k is acc[k].

b[] and c[] represent the 16-bit input mantissa vectors $\bar{b}$ and $\bar{c}$, where each b_k is b[k] and each c_k is c[k].

Each of the input vectors must begin at a word-aligned address.

length is the number of elements in each of the vectors.

acc_shr is the signed arithmetic right-shift applied to the accumulators a_k prior to accumulation.

bc_sat is the unsigned arithmetic right-shift applied to the product of b_k and c_k before accumulation.

Operation Performed

$$v_k \leftarrow round(sat_{16}(b_k \cdot c_k \cdot 2^{-bc\_sat}))$$
$$\hat{a}_k \leftarrow sat_{16}(a_k \cdot 2^{-acc\_shr})$$
$$a_k \leftarrow sat_{16}(\hat{a}_k + v_k)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If inputs $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, and input $\bar{a}$ is the accumulator BFP vector $\bar{a} \cdot 2^{a\_exp}$, then the output values of $\bar{a}$ have the exponent $2^{a\_exp + acc\_shr}$.

For accumulation to make sense mathematically, bc_sat must be chosen such that $a\_exp + acc\_shr = b\_exp + c\_exp + bc\_sat$.

The function *vect_complex_s16_macc_prepare()* can be used to obtain values for a_exp, acc_shr and bc_sat based on the input exponents a_exp, b_exp and c_exp and the input headrooms a_hr, b_hr and c_hr.

See also:

vect_s16_macc_prepare

Parameters

- `acc` − **[inout]** Accumulator $\bar{a}$
- `b` − **[in]** Input vector $\bar{b}$
- `c` − **[in]** Input vector $\bar{c}$
- `length` − **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- `acc_shr` − **[in]** Signed arithmetic right-shift applied to accumulator elements.
- `bc_sat` − **[in]** Unsigned arithmetic right-shift applied to the products of elements b_k and c_k

Throws ET_LOAD_STORE
Raised if `acc`, `b` or `c` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$

headroom_t **vect_s16_nmacc**(int16_t acc[], const int16_t b[], const int16_t c[], const unsigned length, const *right_shift_t* acc_shr, const *right_shift_t* bc_sat)

Multiply one 16-bit vector element-wise by another, and subtract the result from an accumulator.

`acc[]` represents the 16-bit accumulator mantissa vector $\bar{a}$. Each a_k is `acc[k]`.

`b[]` and `c[]` represent the 16-bit input mantissa vectors $\bar{b}$ and $\bar{c}$, where each b_k is `b[k]` and each c_k is `c[k]`.

Each of the input vectors must begin at a word-aligned address.

`length` is the number of elements in each of the vectors.

`acc_shr` is the signed arithmetic right-shift applied to the accumulators a_k prior to accumulation.

`bc_sat` is the unsigned arithmetic right-shift applied to the product of b_k and c_k before accumulation.

Operation Performed

$$v_k \leftarrow round(sat_{16}(b_k \cdot c_k \cdot 2^{-bc\_sat}))$$
$$\hat{a}_k \leftarrow sat_{16}(a_k \cdot 2^{-acc\_shr})$$
$$a_k \leftarrow sat_{16}(\hat{a}_k - v_k)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If inputs $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, and input $\bar{a}$ is the accumulator BFP vector $\bar{a} \cdot 2^{a\_exp}$, then the output values of $\bar{a}$ have the exponent $2^{a\_exp+acc\_shr}$.

For accumulation to make sense mathematically, bc_sat must be chosen such that $a\_exp + acc\_shr = b\_exp + c\_exp + bc\_sat$.

The function *vect_complex_s16_nmacc_prepare()* can be used to obtain values for a_exp, acc_shr and bc_sat based on the input exponents a_exp, b_exp and c_exp and the input head-rooms a_hr, b_hr and c_hr.

See also:

vect_s16_nmacc_prepare

Parameters

- **acc** – **[inout]** Accumulator $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **c** – **[in]** Input vector $\bar{c}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **acc_shr** – **[in]** Signed arithmetic right-shift applied to accumulator elements.
- **bc_sat** – **[in]** Unsigned arithmetic right-shift applied to the products of elements b_k and c_k

Throws ET_LOAD_STORE
Raised if `acc`, `b` or `c` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$

headroom_t **vect_s16_mul**(int16_t a[], const int16_t b[], const int16_t c[], const unsigned length, const *right_shift_t* a_shr)

Multiply two 16-bit vectors together element-wise.

`a[]`, `b[]` and `c[]` represent the 16-bit vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on `b[]` or `c[]`.

`length` is the number of elements in each of the vectors.

`a_shr` is an unsigned arithmetic right-shift applied to the 32-bit accumulators holding the penultimate results.

Operation Performed

$$a'_k \leftarrow b_k \cdot c_k$$
$$a_k \leftarrow sat_{16}(round(a'_k \cdot 2^{-a\_shr}))$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + c\_exp + a\_shr$.

The function *vect_s16_mul_prepare()* can be used to obtain values for a_exp and a_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **c** – **[in]** Input vector $\bar{c}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **a_shr** – **[in]** Right-shift appled to 32-bit products

Throws ET_LOAD_STORE
Raised if a, b or c is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of vector $\bar{a}$

headroom_t **vect_s16_rect**(int16_t a[], const int16_t b[], const unsigned length)

Rectify the elements of a 16-bit vector.

Rectification ensures that all outputs are non-negative, changing negative values to 0.

a[] and b[] represent the 16-bit vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[].

length is the number of elements in each of the vectors.

Each output element a[k] is set to the value of the corresponding input element b[k] if it is positive, and a[k] is set to zero otherwise.

Operation Performed

$$a_k \leftarrow \begin{cases} b_k & b_k > 0 \\ 0 & b_k \le 0 \end{cases}$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the output vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE
Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$.

headroom_t **vect_s16_scale**(int16_t a[], const int16_t b[], const unsigned length, const int16_t c, const *right_shift_t* a_shr)

Multiply a 16-bit vector by a 16-bit scalar.

`a[]` and `b[]` represent the 16-bit vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on `b[]`.

`length` is the number of elements in each of the vectors.

`c` is the 16-bit scalar c by which elements of $\bar{b}$ are multiplied.

`a_shr` is an unsigned arithmetic right-shift applied to the 32-bit accumulators holding the penultimate results.

Operation Performed

$$a'_k \leftarrow b_k \cdot c$$
$$a_k \leftarrow sat_{16}(round(a'_k \cdot 2^{-a\_shr}))$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$ and c is the mantissa of floating-point value $c \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + c\_exp + a\_shr$.

The function *vect_s16_scale_prepare()* can be used to obtain values for a_exp and a_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Parameters

- **a** − **[out]** Output vector $\bar{a}$

- **b** − **[in]** Input vector $\bar{b}$

- **c** − **[in]** Input vector $\bar{c}$

- **length** − **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$

- **a_shr** − **[in]** Right-shift appled to 32-bit products

Throws ET_LOAD_STORE
Raised if `a` or `b` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of vector $\bar{a}$

void **vect_s16_set**(int16_t a[], const int16_t b, const unsigned length)

Set all elements of a 16-bit vector to the specified value.

`a[]` represents the 16-bit vector $\bar{a}$. It must begin at a word-aligned address.

`b` is the value elements of $\bar{a}$ are set to.

`length` is the number of elements in `a[]`.

Operation Performed

$$a_k \leftarrow b$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If b is the mantissa of floating-point value $b \cdot 2^{b\_exp}$, then the output vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Parameters

- **a** − **[out]** Output vector $\bar{a}$
- **b** − **[in]** Input value b
- **length** − **[in]** Number of elements in vector $\bar{a}$

Throws ET_LOAD_STORE
Raised if `a` is not word-aligned (See *Note: Vector Alignment*)

headroom_t **vect_s16_shl**(int16_t a[], const int16_t b[], const unsigned length, const *left_shift_t* b_shl)

Left-shift the elements of a 16-bit vector by a specified number of bits.

`a[]` and `b[]` represent the 16-bit vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on `b[]`.

`length` is the number of elements in vectors $\bar{a}$ and $\bar{b}$.

`b_shl` is the signed arithmetic left-shift applied to each element of $\bar{b}$.

Operation Performed

$$a_k \leftarrow sat_{16}(\lfloor b_k \cdot 2^{b\_shl} \rfloor)$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $\bar{a} = \bar{b} \cdot 2^{b\_shl}$ and $a\_exp = b\_exp$.

Parameters

- **a** − **[out]** Output vector $\bar{a}$
- **b** − **[in]** Input vector $\bar{b}$
- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- **b_shl** − **[in]** Arithmetic left-shift applied to elements of $\bar{b}$

Throws ET_LOAD_STORE
Raised if `a` or `b` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of output vector $\bar{a}$

headroom_t **vect_s16_shr**(int16_t a[], const int16_t b[], const unsigned length, const *right_shift_t* b_shr)

Right-shift the elements of a 16-bit vector by a specified number of bits.

`a[]` and `b[]` represent the 16-bit vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on `b[]`.

`length` is the number of elements in vectors $\bar{a}$ and $\bar{b}$.

`b_shr` is the signed arithmetic right-shift applied to each element of $\bar{b}$.

Operation Performed

$$a_k \leftarrow sat_{16}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $\bar{a} = \bar{b} \cdot 2^{-b\_shr}$ and $a\_exp = b\_exp$.

Parameters

- **a** − **[out]** Output vector $\bar{a}$

- **b** − **[in]** Input vector $\bar{b}$

- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

- **b_shr** − **[in]** Arithmetic right-shift applied to elements of $\bar{b}$

Throws ET_LOAD_STORE
Raised if `a` or `b` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of output vector $\bar{a}$

headroom_t **vect_s16_sqrt**(int16_t a[], const int16_t b[], const unsigned length, const *right_shift_t* b_shr, const unsigned depth)

Compute the square roots of elements of a 16-bit vector.

`a[]` and `b[]` represent the 16-bit vectors $\bar{a}$ and $\bar{b}$ respectively. Each vector must begin at a word-aligned address. This operation can be performed safely in-place on `b[]`.

`length` is the number of elements in each of the vectors.

`b_shr` is the signed arithmetic right-shift applied to elements of $\bar{b}$.

`depth` is the number of most significant bits to calculate of each a_k. For example, a `depth` value of 8 will only compute the 8 most significant byte of the result, with the remaining byte as 0. The maximum value for this parameter is VECT_SQRT_S16_MAX_DEPTH (31). The time cost of this operation is approximately proportional to the number of bits computed.

Operation Performed

$$b'_k \leftarrow sat_{16}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$a_k \leftarrow \begin{cases} \sqrt{b'_k} & b'_k >= 0 \\ 0 & otherwise \end{cases}$$
$$\text{for } k \in 0 \dots (length - 1)$$

where $\sqrt{\cdot}$ computes the most significant $depth$ bits of the square root.

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = (b\_exp + b\_shr - 14)/2$.

Note that because exponents must be integers, that means $b\_exp + b\_shr$ **must be even**.

The function *vect_s16_sqrt_prepare()* can be used to obtain values for a_exp and b_shr based on the input exponent b_exp and headroom b_hr.

Notes

- This function assumes roots are real. Negative input elements will result in corresponding outputs of 0.

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- **b_shr** – **[in]** Right-shift appled to $\bar{b}$
- **depth** – **[in]** Number of bits of each output value to compute

Throws ET_LOAD_STORE

Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of output vector $\bar{a}$

headroom_t **vect_s16_sub** (int16_t a[], const int16_t b[], const int16_t c[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Subtract one 16-bit BFP vector from another.

a[], b[] and c[] represent the 16-bit vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[] or c[].

length is the number of elements in each of the vectors.

b_shr and c_shr are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k = sat_{16}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k = sat_{16}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$a_k \leftarrow sat_{16}(b'_k - c'_k)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$.

In this case, b_shr and c_shr **must** be chosen so that $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$. Adding or subtracting mantissas only makes sense if they are associated with the same exponent.

The function *vect_s16_sub_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

See also:

vect_s16_sub_prepare

Parameters

- **a** − **[out]** Output vector $\bar{a}$
- **b** − **[in]** Input vector $\bar{b}$
- **c** − **[in]** Input vector $\bar{c}$
- **length** − **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **b_shr** − **[in]** Right-shift appled to $\bar{b}$
- **c_shr** − **[in]** Right-shift appled to $\bar{c}$

Throws ET_LOAD_STORE
Raised if a, b or c is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$.

int32_t **vect_s16_sum**(const int16_t b[], const unsigned length)

Get the sum of elements of a 16-bit vector.

b[] represents the 16-bit vector $\bar{b}$. b[] must begin at a word-aligned address.

length is the number of elements in $\bar{b}$.

Operation Performed

$$a \leftarrow \sum_{k=0}^{length-1} b_k$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the returned value a is the 32-bit mantissa of floating-point value $a \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Parameters

- **b** − **[in]** Input vector $\bar{b}$
- **length** − **[in]** Number of elements in $\bar{b}$

Throws ET_LOAD_STORE
Raised if b is not word-aligned (See *Note: Vector Alignment*)

Returns
The 32-bit sum a

unsigned **chunk_s16_accumulate**(*split_acc_s32_t* *acc, const int16_t b[VPU_INT16_EPV], const *right_shift_t* b_shr, const unsigned vpu_ctrl)

Accumulate a 16-bit vector chunk into a 32-bit accumulator chunk.

16-bit vector chunk $\bar{b}$ is shifted and accumulated into 32-bit accumulator vector chunk $\bar{a}$ (`acc`). This function is used for efficiently accumulating multiple (possibly many) 16-bit vectors together.

The accumulator vector $\bar{a}$ stores its elements across two 16-bit vector chunks, which corresponds to how the accumulators are stored internally across VPU registers `vD` and `vR`. See *split_acc_s32_t* for details about the accumulator structure.

The signed arithmetic right-shift `b_shr` is applied to $\bar{b}$ prior to being accumulated into $\bar{a}$. When $\bar{b}$ and $\bar{a}$, are the mantissas of block floating point vectors, using `b_shr` allows those vectors to have different exponents. This is also important when this function is to be called periodically where each $\bar{b}$ may have a different exponent.

`b_shr` must meet the condition `-14 <= b_shr <= 14` or the behavior of this function is undefined.

Operation Performed

$$a_k \leftarrow a_k + floor(\frac{b_k}{2^{-\text{b\_shr}}})$$

The input `vpu_ctrl` tracks the VPU's control register state during accumulation. In particular, it is used for keeping track of the headroom of the accumulator vector $\bar{a}$. When beginning a sequence of accumulation calls, the value passed in should be initialized to `VPU_INT16_CTRL_INIT`. On completion, this function returns the updated VPU control register state, which should be passed in as `vpu_ctrl` on the next accumulation call.

VPU Control Value

The idea is that each call to this function processes only a single 'chunk' (in 16-bit mode, a 16-element block) at a time, but the caller usually wants to know the headroom of a whole vector, which may comprise many such chunks. So `vpu_ctrl` is a value which persists through each of these calls to track the whole vector.

Once all chunks have been accumulated, the `VPU_INT16_HEADROOM_FROM_CTRL()` macro can be used to get the headroom of the accumulator vector. Note that this will produce a maximum value of `15`.

If many vector chunks $\bar{b}$ are accumulated into the same accumulators (when using block floating-point, it may be only a few accumulations if the exponent associated with $\bar{b}$ is significantly larger than that associated with $\bar{a}$), saturation becomes possible.

Accumulating Many Values

When saturation is possible, the user must monitor the headroom of $\bar{a}$ (using the returned value and `VPU_INT16_HEADROOM_FROM_CTRL()`) to detect when there is no further headroom. As long as there is at least 1 bit of headroom, a call to this function cannot saturate.

Typically, when using block floating-point, this will be handled by:

- Converting $\bar{a}$ to a standard vector of `int32_t` using *vect_s32_merge_accs()*
- Right-shift the values of $\bar{a}$ using *vect_s32_shr()*
- Increment the exponent associated with $\bar{a}$ by the same amount right-shifted
- Convert $\bar{a}$ back into the split accumulator format using *vect_s32_split_accs()*

When accumulating, setting `b_shr` to the exponent associated with $\bar{b}$ minus the exponent associated with $\bar{a}$ will automatically adjust for the new exponent of $\bar{a}$.

Parameters

- `acc` – **[inout]** b
- `b` – **[in]** v
- `b_shr` – **[in]** v
- `vpu_ctrl` – **[in]** e

Throws ET_LOAD_STORE

Raised if `acc` or `b` is not word-aligned (See *Note: Vector Alignment*)

Returns

Current state of VPU control register.

void **vect_s16_to_vect_s32**(int32_t a[], const int16_t b[], const unsigned length)

Convert a 16-bit vector to a 32-bit vector.

`a[]` represents the 32-bit output vector $\bar{a}$.

`b[]` represents the 16-bit input vector $\bar{b}$.

Each vector must begin at a word-aligned address.

`length` is the number of elements in each of the vectors.

Operation Performed

$$a_k \leftarrow b_k \cdot 2^8$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the 32-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$. If $a\_exp = b\_exp - 8$, then this operation has effectively not changed the values represented.

Notes

- The multiplication by 2^8 is an artifact of the VPU's behavior. It turns out to be significantly more efficient to include the factor of 2^8. If this is unwanted, *vect_s32_shr()* can be used with a `b_shr` value of 8 to remove the scaling afterwards.
- The headroom of output vector $\bar{a}$ is not returned by this function. The headroom of the output is always 8 bits greater than the headroom of the input.

Parameters

- `a` – **[out]** 32-bit output vector $\bar{a}$
- `b` – **[in]** 16-bit input vector $\bar{b}$
- `length` – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE

Raised if `a` or `b` is not word-aligned (See *Note: Vector Alignment*)

void **vect_s16_extract_high_byte**(int8_t a[], const int16_t b[], const unsigned len)

Extract an 8-bit vector containing the most significant byte of a 16-bit vector.

This is a utility function used, for example, in optimizing mixed-width products. The most significant byte of each element is extracted (without rounding or saturation) and inserted into the output vector.

See also:

vect_s16_extract_low_byte

Parameters

- **a** – **[out]** 8-bit output vector $\bar{a}$
- **b** – **[in]** 16-bit input vector $\bar{b}$
- **len** – **[in]** The number of elements in $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE
Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

void **vect_s16_extract_low_byte**(int8_t a[], const int16_t b[], const unsigned len)

Extract an 8-bit vector containing the least significant byte of a 16-bit vector.

This is a utility function used, for example, in optimizing mixed-width products. The least significant byte of each element is extracted (without rounding or saturation) and inserted into the output vector.

See also:

vect_s16_extract_high_byte

Parameters

- **a** – **[out]** 8-bit output vector $\bar{a}$
- **b** – **[in]** 16-bit input vector $\bar{b}$
- **len** – **[in]** The number of elements in $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE
Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

4.7.4 32-Bit Vector API

group **vect_s32_api**

Defines

VECT_SQRT_S32_MAX_DEPTH

Maximum bit-depth that can be calculated by *vect_s32_sqrt()*.

See also:

vect_s32_sqrt

Enums

enum **pad_mode_e**

Supported padding modes for convolutions in "same" mode.

See also:

vect_s32_convolve_same(), *bfp_s32_convolve_same()*

Values:

enumerator **PAD_MODE_REFLECT**

Vector is reflected at its boundaries, such that

$$\tilde{x}_i \begin{cases} x_{-i} & i < 0 \\ x_{2N-2-i} & i \geq N \\ x_i & otherwise \end{cases}$$

For example, if the length N of input vector $\bar{x}$ is 7 and the order K of the filter is 5, then

$$\bar{x} = [x_0, x_1, x_2, x_3, x_4, x_5, x_6]$$

$$\tilde{x} = [x_2, x_1, x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_5, x_4]$$

Note that by convention the first element of $\tilde{x}$ is considered to be at index $-P$, where $P = \lfloor K/2 \rfloor$.

enumerator **PAD_MODE_EXTEND**

Vector is padded using the value of the bounding elements.

$$\tilde{x}_i \begin{cases} x_0 & i < 0 \\ x_{N-1} & i \geq N \\ x_i & otherwise \end{cases}$$

For example, if the length N of input vector $\bar{x}$ is 7 and the order K of the filter is 5, then

$$\bar{x} = [x_0, x_1, x_2, x_3, x_4, x_5, x_6]$$

$$\tilde{x} = [x_0, x_0, x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_6, x_6]$$

Note that by convention the first element of $\tilde{x}$ is considered to be at index $-P$, where $P = \lfloor K/2 \rfloor$.

enumerator **PAD_MODE_ZERO**

Vector is padded with zeroes.

$$\tilde{x}_i \begin{cases} 0 & i < 0 \\ 0 & i \geq N \\ x_i & otherwise \end{cases}$$

For example, if the length N of input vector $\bar{x}$ is 7 and the order K of the filter is 5, then

$$\bar{x} = [x_0, x_1, x_2, x_3, x_4, x_5, x_6]$$

$$\tilde{x} = [0, 0, x_0, x_1, x_2, x_3, x_4, x_5, x_6, 0, 0]$$

Note that by convention the first element of $\tilde{x}$ is considered to be at index $-P$, where $P = \lfloor K/2 \rfloor$.

Functions

headroom_t **vect_s32_copy**(int32_t a[], const int32_t b[], const unsigned length)

Copy one 32-bit vector to another.

This function is effectively a constrained version of `memcpy`.

With the constraints below met, this function should be modestly faster than `memcpy`.

`a[]` is the output vector to which elements are copied.

`b[]` is the input vector from which elements are copied.

`a` and `b` each must begin at a word-aligned address.

`length` is the number of elements to be copied. `length` must be a multiple of 8.

Operation Performed

$$a_k \leftarrow b_k$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Parameters

- **a** − **[out]** Output vector $\bar{a}$
- **b** − **[in]** Input vector $\bar{b}$
- **length** − **[in]** Number of elements in $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE

Raised if `a` or `b` is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of output vector $\bar{a}$

headroom_t **vect_s32_abs**(int32_t a[], const int32_t b[], const unsigned length)

Compute the element-wise absolute value of a 32-bit vector.

`a[]` and `b[]` represent the 32-bit vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on `b[]`.

`length` is the number of elements in each of the vectors.

Operation Performed

$$a_k \leftarrow sat_{32}(|b_k|)$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the output vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE

Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of the output vector $\bar{a}$.

int64_t **vect_s32_abs_sum**(const int32_t b[], const unsigned length)

Compute the sum of the absolute values of elements of a 32-bit vector.

b[] represents the 32-bit mantissa vector $\bar{b}$. b[] must begin at a word-aligned address.

length is the number of elements in $\bar{b}$.

Operation Performed

$$\sum_{k=0}^{length-1} sat_{32}(|b_k|)$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the returned value a is the 64-bit mantissa of floating-point value $a \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Additional Details

Internally the sum accumulates into 8 separate 40-bit accumulators. These accumulators apply symmetric 40-bit saturation logic (with bounds $\pm(2^{39} - 1)$) with each added element. At the end, the 8 accumulators are summed together into the 64-bit value a which is returned by this function. No saturation logic is applied at this final step.

Because symmetric 32-bit saturation logic is applied when computing the absolute value, in the corner case where each element is INT32_MIN, each of the 8 accumulators can accumulate 256 elements before saturation is possible. Therefore, with b_hr bits of headroom, no saturation of intermediate results is possible with fewer than 2^{11+b_hr} elements in $\bar{b}$.

If the length of $\bar{b}$ is greater than 2^{11+b_hr}, the sum can be computed piece-wise in several calls to this function, with the partial results summed in user code.

Parameters

- **b** – **[in]** Input vector $\bar{b}$

- **length** − **[in]** Number of elements in $\bar{b}$

Throws ET_LOAD_STORE
> Raised if `b` is not word-aligned (See *Note: Vector Alignment*)

Returns
> The 64-bit sum a

headroom_t **vect_s32_add** (int32_t a[], const int32_t b[], const int32_t c[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Add together two 32-bit vectors.

`a[]`, `b[]` and `c[]` represent the 32-bit mantissa vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on `b[]` or `c[]`.

`length` is the number of elements in each of the vectors.

`b_shr` and `c_shr` are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k = sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k = sat_{32}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$a_k \leftarrow sat_{32}(b'_k + c'_k)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$.

In this case, b_shr and c_shr **must** be chosen so that $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$. Adding or subtracting mantissas only makes sense if they are associated with the same exponent.

The function *vect_s32_add_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

See also:

vect_s32_add_prepare()

Parameters
- **a** − **[out]** Output vector $\bar{a}$
- **b** − **[in]** Input vector $\bar{b}$
- **c** − **[in]** Input vector $\bar{c}$
- **length** − **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **b_shr** − **[in]** Right-shift appled to $\bar{b}$
- **c_shr** − **[in]** Right-shift appled to $\bar{c}$

Throws ET_LOAD_STORE
> Raised if `a`, `b` or `c` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$.

$headroom\_t$ **vect_s32_add_scalar**(int32_t a[], const int32_t b[], const int32_t c, const unsigned length, const $right\_shift\_t$ b_shr)

Add a scalar to a 32-bit vector.

a[], b[] represent the 32-bit mantissa vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[].

c is the scalar c to be added to each element of $\bar{b}$.

length is the number of elements in each of the vectors.

b_shr is the signed arithmetic right-shift applied to each element of $\bar{b}$.

Operation Performed

$$b'_k = sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$a_k \leftarrow sat_{32}(b'_k + c)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If elements of $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, and c is the mantissa of floating-point value $c \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$.

In this case, b_shr and c_shr **must** be chosen so that $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$. Adding or subtracting mantissas only makes sense if they are associated with the same exponent.

The function *vect_s32_add_scalar_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Note that c_shr is an output of *vect_s32_add_scalar_prepare()*, but is not a parameter to this function. The c_shr produced by *vect_s32_add_scalar_prepare()* is to be applied by the user, and the result passed as input c.

See also:

vect_s32_add_scalar_prepare()

Parameters

- **a** − **[out]** Output vector $\bar{a}$
- **b** − **[in]** Input vector $\bar{b}$
- **c** − **[in]** Input scalar c
- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- **b_shr** − **[in]** Right-shift appled to $\bar{b}$

Throws ET_LOAD_STORE
Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$.

unsigned **vect_s32_argmax**(const int32_t b[], const unsigned length)

>   Obtain the array index of the maximum element of a 32-bit vector.

>   b[] represents the 32-bit input vector $\bar{b}$. It must begin at a word-aligned address.

>   length is the number of elements in $\bar{b}$.

>   **Operation Performed**

$$a \leftarrow argmax_k\{b_k\}$$
$$\text{for } k \in 0 ... (length - 1)$$

>   >   **Parameters**

>   >   >   - **b** − **[in]** Input vector $\bar{b}$

>   >   >   - **length** − **[in]** Number of elemetns in $\bar{b}$

>   >   **Throws ET_LOAD_STORE**
>   >   >   Raised if b is not word-aligned (See *Note: Vector Alignment*)

>   >   **Returns**
>   >   > a, the index of the maximum element of vector $\bar{b}$. If there is a tie for the maximum value, the lowest tying index is returned.

unsigned **vect_s32_argmin**(const int32_t b[], const unsigned length)

>   Obtain the array index of the minimum element of a 32-bit vector.

>   b[] represents the 32-bit input vector $\bar{b}$. It must begin at a word-aligned address.

>   length is the number of elements in $\bar{b}$.

>   **Operation Performed**

$$a \leftarrow argmin_k\{b_k\}$$
$$\text{for } k \in 0 ... (length - 1)$$

>   >   **Parameters**

>   >   >   - **b** − **[in]** Input vector $\bar{b}$

>   >   >   - **length** − **[in]** Number of elemetns in $\bar{b}$

>   >   **Throws ET_LOAD_STORE**
>   >   >   Raised if b is not word-aligned (See *Note: Vector Alignment*)

>   >   **Returns**
>   >   > a, the index of the minimum element of vector $\bar{b}$. If there is a tie for the minimum value, the lowest tying index is returned.

headroom_t **vect_s32_clip**(int32_t a[], const int32_t b[], const unsigned length, const int32_t lower_bound, const int32_t upper_bound, const *right_shift_t* b_shr)

>   Clamp the elements of a 32-bit vector to a specified range.

>   a[] and b[] represent the 32-bit vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[].

>   length is the number of elements in each of the vectors.

>   lower_bound and upper_bound are the lower and upper bounds of the clipping range respectively. These bounds are checked for each element of $\bar{b}$ only *after* b_shr is applied.

b_shr is the signed arithmetic right-shift applied to elements of $\bar{b}$ *before* being compared to the upper and lower bounds.

If $\bar{b}$ are the mantissas for a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the exponent a_exp of the output BFP vector $\bar{a} \cdot 2^{a\_exp}$ is given by $a\_exp = b\_exp + b\_shr$.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$

$$a_k \leftarrow \begin{cases} lower\_bound & b'_k \leq lower\_bound \\ upper\_bound & b'_k \geq upper\_bound \\ b'_k & otherwise \end{cases}$$

$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the output vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + b\_shr$.

Parameters

- **a** − **[out]** Output vector $\bar{a}$
- **b** − **[in]** Input vector $\bar{b}$
- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- **lower_bound** − **[in]** Lower bound of clipping range
- **upper_bound** − **[in]** Upper bound of clipping range
- **b_shr** − **[in]** Arithmetic right-shift applied to elements of $\bar{b}$ prior to clipping

Throws ET_LOAD_STORE
Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of output vector $\bar{a}$

int64_t **vect_s32_dot**(const int32_t b[], const int32_t c[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Compute the inner product between two 32-bit vectors.

b[] and c[] represent the 32-bit mantissa vectors $\bar{b}$ and $\bar{c}$ respectively. Each must begin at a word-aligned address.

length is the number of elements in each of the vectors.

b_shr and c_shr are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k \leftarrow sat_{32}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$a \leftarrow \sum_{k=0}^{length-1} \left(round(b'_k \cdot c'_k \cdot 2^{-30}) \right)$$

where a is returned

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the mantissas of the BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then result a is the 64-bit mantissa of the result $a \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + c\_exp + b\_shr + c\_shr + 30$.

If needed, the bit-depth of a can then be reduced to 32 bits to get a new result $a' \cdot 2^{a\_exp'}$ where $a' = a \cdot 2^{-a\_shr}$ and $a\_exp' = a\_exp + a\_shr$.

The function *vect_s32_dot_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Additional Details

The 30-bit rounding right-shift applied to each of the 64-bit products $b_k \cdot c_k$ is a feature of the hardware and cannot be avoided. As such, if the input vectors $\bar{b}$ and $\bar{c}$ together have too much headroom (i.e. $b\_hr + c\_hr$), the sum may effectively vanish. To avoid this situation, negative values of `b_shr` and `c_shr` may be used (with the stipulation that $b\_shr \geq -b\_hr$ and $c\_shr \geq -c\_hr$ if saturation of b'_k and c'_k is to be avoided). The less headroom b'_k and c'_k have, the greater the precision of the final result.

Internally, each product $(b'_k \cdot c'_k \cdot 2^{-30})$ accumulates into one of eight 40-bit accumulators (which are all used simultaneously) which apply symmetric 40-bit saturation logic (with bounds $\approx 2^{39}$) with each value added. The saturating arithmetic employed is *not associative* and no indication is given if saturation occurs at an intermediate step. To avoid satuation errors, `length` should be no greater than $2^{10+b\_hr+c\_hr}$, where b_hr and c_hr are the headroom of $\bar{b}$ and $\bar{c}$ respectively.

If the caller's mantissa vectors are longer than that, the full inner product can be found by calling this function multiple times for partial inner products on sub-sequences of the input vectors, and adding the results in user code.

In many situations the caller may have *a priori* knowledge that saturation is impossible (or very nearly so), in which case this guideline may be disregarded. However, such situations are application-specific and are well beyond the scope of this documentation, and as such are left to the user's discretion.

Parameters

- **b** − **[in]** Input vector $\bar{b}$
- **c** − **[in]** Input vector $\bar{c}$
- **length** − **[in]** Number of elements in vectors $\bar{b}$ and $\bar{c}$
- **b_shr** − **[in]** Right-shift appled to $\bar{b}$
- **c_shr** − **[in]** Right-shift appled to $\bar{c}$

Throws ET_LOAD_STORE
Raised if `b` or `c` is not word-aligned (See *Note: Vector Alignment*)

Returns
The inner product of vectors $\bar{b}$ and $\bar{c}$, scaled as indicated above.

int64_t **vect_s32_energy**(const int32_t b[], const unsigned length, const *right_shift_t* b_shr)
Calculate the energy (sum of squares of elements) of a 32-bit vector.

`b[]` represents the 32-bit mantissa vector $\bar{b}$. `b[]` must begin at a word-aligned address.

`length` is the number of elements in $\bar{b}$.

`b_shr` is the signed arithmetic right-shift applied to elements of $\bar{b}$.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$

$$a \leftarrow \sum_{k=0}^{length-1} round((b'_k)^2 \cdot 2^{-30})$$

Block Floating-Point

If $\bar{b}$ are the mantissas of the BFP vector $\bar{b} \cdot 2^{b\_exp}$, then floating-point result is $a \cdot 2^{a\_exp}$, where the 64-bit mantissa a is returned by this function, and $a\_exp = 30 + 2 \cdot (b\_exp + b\_shr)$.

The function *vect_s32_energy_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Additional Details

The 30-bit rounding right-shift applied to each element of the 64-bit products $(b'_k)^2$ is a feature of the hardware and cannot be avoided. As such, if the input vector $\bar{b}$ has too much headroom (i.e. $2 \cdot b\_hr$), the sum may effectively vanish. To avoid this situation, negative values of `b_shr` may be used (with the stipulation that $b\_shr \geq -b\_hr$ if satuartion of b'_k is to be avoided). The less headroom b'_k has, the greater the precision of the final result.

Internally, each product $(b'_k)^2 \cdot 2^{-30}$ accumulates into one of eight 40-bit accumulators (which are all used simultaneously) which apply symmetric 40-bit saturation logic (with bounds $\approx 2^{39}$) with each value added. The saturating arithmetic employed is *not associative* and no indication is given if saturation occurs at an intermediate step. To avoid saturation errors, `length` should be no greater than $2^{10+2\cdot b\_hr}$, where b_hr is the headroom of $\bar{b}$.

If the caller's mantissa vector is longer than that, the full result can be found by calling this function multiple times for partial results on sub-sequences of the input, and adding the results in user code.

In many situations the caller may have *a priori* knowledge that saturation is impossible (or very nearly so), in which case this guideline may be disregarded. However, such situations are application-specific and are well beyond the scope of this documentation, and as such are left to the user's discretion.

Parameters

- **b** − **[in]** Input vector $\bar{b}$
- **length** − **[in]** Number of elements in $\bar{b}$
- **b_shr** − **[in]** Right-shift appled to $\bar{b}$

Throws ET_LOAD_STORE
Raised if b is not word-aligned (See *Note: Vector Alignment*)

Returns
64-bit mantissa of vector $\bar{b}$'s energy

headroom_t **vect_s32_headroom**(const int32_t x[], const unsigned length)

Calculate the headroom of a 32-bit vector.

The headroom of an N-bit integer is the number of bits that the integer's value may be left-shifted without any information being lost. Equivalently, it is one less than the number of leading sign bits.

The headroom of an int32_t array is the minimum of the headroom of each of its int32_t elements.

This function efficiently traverses the elements of a[] to determine its headroom.

x[] represents the 32-bit vector $\bar{x}$. x[] must begin at a word-aligned address.

length is the number of elements in x[].

Operation Performed

$$min\{HR_{32}(x_0), HR_{32}(x_1), ..., HR_{32}(x_{length-1})\}$$

See also:

vect_s16_headroom, *vect_complex_s16_headroom*, *vect_complex_s32_headroom*

Parameters

- **x** − **[in]** Input vector $\bar{x}$

- **length** − **[in]** The number of elements in x[]

Throws ET_LOAD_STORE
Raised if x is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of vector $\bar{x}$

headroom_t **vect_s32_inverse**(int32_t a[], const int32_t b[], const unsigned length, const unsigned scale)

Compute the inverse of elements of a 32-bit vector.

a[] and b[] represent the 32-bit mantissa vectors $\bar{a}$ and $\bar{b}$ respectively. Each vector must begin at a word-aligned address. This operation can be performed safely in-place on b[].

length is the number of elements in each of the vectors.

scale is a scaling parameter used to maximize the precision of the result.

Operation Performed

$$a_k \leftarrow \lfloor \frac{2^{scale}}{b_k} \rfloor$$
$$\text{for } k \in 0 ... (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = scale - b\_exp$.

The function *vect_s32_inverse_prepare()* can be used to obtain values for a_exp and $scale$.

vect_s32_inverse_prepare

Parameters

- **a** – **[out]** Output vector $\bar{a}$

- **b** – **[in]** Input vector $\bar{b}$

- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

- **scale** – **[in]** Scale factor applied to dividend when computing inverse

Throws ET_LOAD_STORE
Raised if `a` or `b` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of output vector $\bar{a}$

int32_t **vect_s32_max**(const int32_t b[], const unsigned length)

Find the maximum value in a 32-bit vector.

`b[]` represents the 32-bit vector $\bar{b}$. It must begin at a word-aligned address.

`length` is the number of elements in $\bar{b}$.

Operation Performed

$$max\{x_0, x_1, ..., x_{length-1}\}$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the returned value a is the 32-bit mantissa of floating-point value $a \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Parameters

- **b** – **[in]** Input vector $\bar{b}$

- **length** – **[in]** Number of elements in $\bar{b}$

Throws ET_LOAD_STORE
Raised if `b` is not word-aligned (See *Note: Vector Alignment*)

Returns
Maximum value from $\bar{b}$

headroom_t **vect_s32_max_elementwise**(int32_t a[], const int32_t b[], const int32_t c[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Get the element-wise maximum of two 32-bit vectors.

`a[]`, `b[]` and `c[]` represent the 32-bit mantissa vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on `b[]`, but *not* on `c[]`.

`length` is the number of elements in each of the vectors.

`b_shr` and `c_shr` are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k \leftarrow sat_{32}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$a_k \leftarrow max(b'_k, c'_k)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$.

The function *vect_2vec_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

> **Warning:** For correct operation, this function requires at least 1 bit of headroom in each mantissa vector *after* the shifts have been applied.

Parameters

- **a** − **[out]** Output vector $\bar{a}$
- **b** − **[in]** Input vector $\bar{b}$
- **c** − **[in]** Input vector $\bar{c}$
- **length** − **[in]** Number of elements in vectors $\bar{a}, \bar{b}$ and $\bar{c}$
- **b_shr** − **[in]** Right-shift appled to $\bar{b}$
- **c_shr** − **[in]** Right-shift appled to $\bar{c}$

Throws ET_LOAD_STORE
Raised if a, b or c is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of vector $\bar{a}$

int32_t **vect_s32_min**(const int32_t b[], const unsigned length)
Find the minimum value in a 32-bit vector.

b[] represents the 32-bit vector $\bar{b}$. It must begin at a word-aligned address.

length is the number of elements in $\bar{b}$.

Operation Performed

$$max\{x_0, x_1, ..., x_{length-1}\}$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the returned value a is the 32-bit mantissa of floating-point value $a \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Parameters

- **b** − **[in]** Input vector $\bar{b}$

- **length** − **[in]** Number of elements in $\bar{b}$

Throws ET_LOAD_STORE
 Raised if b is not word-aligned (See *Note: Vector Alignment*)

Returns
 Minimum value from $\bar{b}$

headroom_t **vect_s32_min_elementwise**(int32_t a[], const int32_t b[], const int32_t c[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Get the element-wise minimum of two 32-bit vectors.

a[], b[] and c[] represent the 32-bit mantissa vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[], but *not* on c[].

length is the number of elements in each of the vectors.

b_shr and c_shr are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k \leftarrow sat_{32}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$a_k \leftarrow min(b'_k, c'_k)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$.

The function *vect_2vec_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

> **Warning:** For correct operation, this function requires at least 1 bit of headroom in each mantissa vector *after* the shifts have been applied.

Parameters

- **a** − **[out]** Output vector $\bar{a}$

- **b** − **[in]** Input vector $\bar{b}$

- **c** − **[in]** Input vector $\bar{c}$

- **length** − **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$

- **b_shr** − **[in]** Right-shift appled to $\bar{b}$

- **c_shr** − **[in]** Right-shift appled to $\bar{c}$

Throws ET_LOAD_STORE
 Raised if a, b or c is not word-aligned (See *Note: Vector Alignment*)

Returns
 Headroom of vector $\bar{a}$

headroom_t **vect_s32_mul**(int32_t a[], const int32_t b[], const int32_t c[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Multiply one 32-bit vector element-wise by another.

`a[]`, `b[]` and `c[]` represent the 32-bit mantissa vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on `b[]` or `c[]`.

`length` is the number of elements in each of the vectors.

`b_shr` and `c_shr` are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k \leftarrow sat_{32}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$a_k \leftarrow sat_{32}(round(b'_k \cdot c'_k \cdot 2^{-30}))$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + c\_exp + b\_shr + c\_shr + 30$.

The function *vect_s32_mul_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **c** – **[in]** Input vector $\bar{c}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **b_shr** – **[in]** Right-shift appled to $\bar{b}$
- **c_shr** – **[in]** Right-shift appled to $\bar{c}$

Throws ET_LOAD_STORE
Raised if `a`, `b` or `c` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of vector $\bar{a}$

headroom_t **vect_s32_macc**(int32_t acc[], const int32_t b[], const int32_t c[], const unsigned length, const *right_shift_t* acc_shr, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Multiply one 32-bit vector element-wise by another, and add the result to an accumulator.

`acc[]` represents the 32-bit accumulator mantissa vector $\bar{a}$. Each a_k is `acc[k]`.

`b[]` and `c[]` represent the 32-bit input mantissa vectors $\bar{b}$ and $\bar{c}$, where each b_k is `b[k]` and each c_k is `c[k]`.

Each of the input vectors must begin at a word-aligned address.

`length` is the number of elements in each of the vectors.

acc_shr, b_shr and c_shr are the signed arithmetic right-shifts applied to input elements a_k, b_k and c_k.

Operation Performed

$$\tilde{b}_k \leftarrow sat_{32}(b_k \cdot 2^{-b\_shr})$$
$$\tilde{c}_k \leftarrow sat_{32}(c_k \cdot 2^{-c\_shr})$$
$$\tilde{a}_k \leftarrow sat_{32}(a_k \cdot 2^{-acc\_shr})$$
$$v_k \leftarrow round(sat_{32}(\tilde{b}_k \cdot \tilde{c}_k \cdot 2^{-30}))$$
$$a_k \leftarrow sat_{32}(\tilde{a}_k + v_k)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If inputs $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, and input $\bar{a}$ is the accumulator BFP vector $\bar{a} \cdot 2^{a\_exp}$, then the output values of $\bar{a}$ have the exponent $2^{a\_exp+acc\_shr}$.

For accumulation to make sense mathematically, bc_sat must be chosen such that $a\_exp + acc\_shr = b\_exp + c\_exp + bc\_sat$.

The function *vect_complex_s16_macc_prepare()* can be used to obtain values for a_exp, acc_shr and bc_sat based on the input exponents a_exp, b_exp and c_exp and the input headrooms a_hr, b_hr and c_hr.

See also:

vect_s32_macc_prepare

Parameters

- **acc** − **[inout]** Accumulator $\bar{a}$
- **b** − **[in]** Input vector $\bar{b}$
- **c** − **[in]** Input vector $\bar{c}$
- **length** − **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **acc_shr** − **[in]** Signed arithmetic right-shift applied to accumulator elements.
- **b_shr** − **[in]** Signed arithmetic right-shift applied to elements of $\bar{b}$
- **c_shr** − **[in]** Signed arithmetic right-shift applied to elements of $\bar{c}$

Throws ET_LOAD_STORE
Raised if acc, b or c is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$

headroom_t **vect_s32_nmacc**(int32_t acc[], const int32_t b[], const int32_t c[], const unsigned length, const *right_shift_t* acc_shr, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Multiply one 32-bit vector element-wise by another, and subtract the result from an accumulator.

acc[] represents the 32-bit accumulator mantissa vector $\bar{a}$. Each a_k is acc[k].

b[] and c[] represent the 32-bit input mantissa vectors $\bar{b}$ and $\bar{c}$, where each b_k is b[k] and each c_k is c[k].

Each of the input vectors must begin at a word-aligned address.

`length` is the number of elements in each of the vectors.

`acc_shr`, `b_shr` and `c_shr` are the signed arithmetic right-shifts applied to input elements a_k, b_k and c_k.

Operation Performed

$$\tilde{b}_k \leftarrow sat_{32}(b_k \cdot 2^{-b\_shr})$$
$$\tilde{c}_k \leftarrow sat_{32}(c_k \cdot 2^{-c\_shr})$$
$$\tilde{a}_k \leftarrow sat_{32}(a_k \cdot 2^{-acc\_shr})$$
$$v_k \leftarrow round(sat_{32}(\tilde{b}_k \cdot \tilde{c}_k \cdot 2^{-30}))$$
$$a_k \leftarrow sat_{32}(\tilde{a}_k - v_k)$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If inputs $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, and input $\bar{a}$ is the accumulator BFP vector $\bar{a} \cdot 2^{a\_exp}$, then the output values of $\bar{a}$ have the exponent $2^{a\_exp+acc\_shr}$.

For accumulation to make sense mathematically, bc_sat must be chosen such that $a\_exp + acc\_shr = b\_exp + c\_exp + bc\_sat$.

The function *vect_complex_s16_macc_prepare()* can be used to obtain values for a_exp, acc_shr and bc_sat based on the input exponents a_exp, b_exp and c_exp and the input headrooms a_hr, b_hr and c_hr.

See also:

vect_s32_nmacc_prepare

Parameters

- **acc** – **[inout]** Accumulator $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **c** – **[in]** Input vector $\bar{c}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **acc_shr** – **[in]** Signed arithmetic right-shift applied to accumulator elements.
- **b_shr** – **[in]** Signed arithmetic right-shift applied to elements of $\bar{b}$
- **c_shr** – **[in]** Signed arithmetic right-shift applied to elements of $\bar{c}$

Throws ET_LOAD_STORE
Raised if `acc`, `b` or `c` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$

headroom_t **vect_s32_rect**(int32_t a[], const int32_t b[], const unsigned length)

Rectify the elements of a 32-bit vector.

`a[]` and `b[]` represent the 32-bit mantissa vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on `b[]`.

length is the number of elements in each of the vectors.

Operation Performed

$$a_k \leftarrow \begin{cases} b_k & b_k > 0 \\ 0 & b_k \leq 0 \end{cases}$$

$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the output vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE

Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of the output vector $\bar{a}$

headroom_t **vect_s32_scale**(int32_t a[], const int32_t b[], const unsigned length, const int32_t c, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Multiply a 32-bit vector by a scalar.

a[] and b[] represent the 32-bit mantissa vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[].

length is the number of elements in each of the vectors.

c is the 32-bit scalar c by which each element of $\bar{b}$ is multiplied.

b_shr and c_shr are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and to c.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$a_k \leftarrow sat_{32}(round(c \cdot b'_k \cdot 2^{-30}))$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$ and c is the mantissa of floating-point value $c \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + c\_exp + b\_shr + c\_shr + 30$.

The function *vect_s32_scale_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- **c** – **[in]** Scalar to be multiplied by elements of $\bar{b}$
- **b_shr** – **[in]** Right-shift appled to $\bar{b}$
- **c_shr** – **[in]** Right-shift applied to c

Throws ET_LOAD_STORE

Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of output vector $\bar{a}$

void **vect_s32_set**(int32_t a[], const int32_t b, const unsigned length)

Set all elements of a 32-bit vector to the specified value.

a[] represents the 32-bit output vector $\bar{a}$. a[] must begin at a word-aligned address.

b is the new value to set each element of $\bar{a}$ to.

Operation Performed

$$a_k \leftarrow b$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If b is the mantissa of floating-point value $b \cdot 2^{b\_exp}$, then the output vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** New value for the elements of $\bar{a}$
- **length** – **[in]** Number of elements in $\bar{a}$

Throws ET_LOAD_STORE

Raised if a is not word-aligned (See *Note: Vector Alignment*)

headroom_t **vect_s32_shl**(int32_t a[], const int32_t b[], const unsigned length, const *left_shift_t* b_shl)

Left-shift the elements of a 32-bit vector by a specified number of bits.

a[] and b[] represent the 32-bit vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[].

length is the number of elements in vectors $\bar{a}$ and $\bar{b}$.

b_shl is the signed arithmetic left-shift applied to each element of $\bar{b}$.

Operation Performed

$$a_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{b\_shl} \rfloor)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $\bar{a} = \bar{b} \cdot 2^{b\_shl}$ and $a\_exp = b\_exp$.

Parameters

- **a** − **[out]** Output vector $\bar{a}$
- **b** − **[in]** Input vector $\bar{b}$
- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- **b_shl** − **[in]** Arithmetic left-shift applied to elements of $\bar{b}$

Throws ET_LOAD_STORE
Raised if `a` or `b` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of output vector $\bar{a}$

headroom_t **vect_s32_shr**(int32_t a[], const int32_t b[], const unsigned length, const *right_shift_t* b_shr)

Right-shift the elements of a 32-bit vector by a specified number of bits.

`a[]` and `b[]` represent the 32-bit vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on `b[]`.

`length` is the number of elements in vectors $\bar{a}$ and $\bar{b}$.

`b_shr` is the signed arithmetic right-shift applied to each element of $\bar{b}$.

Operation Performed

$$a_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $\bar{a} = \bar{b} \cdot 2^{-b\_shr}$ and $a\_exp = b\_exp$.

Parameters

- **a** − **[out]** Output vector $\bar{a}$
- **b** − **[in]** Input vector $\bar{b}$
- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- **b_shr** − **[in]** Arithmetic right-shift applied to elements of $\bar{b}$

Throws ET_LOAD_STORE
Raised if `a` or `b` is not word-aligned (See *Note: Vector Alignment*)

Returns

    Headroom of output vector $\bar{a}$

headroom_t **vect_s32_sqrt**(int32_t a[], const int32_t b[], const unsigned length, const *right_shift_t* b_shr, const unsigned depth)

Compute the square root of elements of a 32-bit vector.

a[] and b[] represent the 32-bit mantissa vectors $\bar{a}$ and $\bar{b}$ respectively. Each vector must begin at a word-aligned address. This operation can be performed safely in-place on b[].

length is the number of elements in each of the vectors.

b_shr is the signed arithmetic right-shift applied to elements of $\bar{b}$.

depth is the number of most significant bits to calculate of each a_k. For example, a depth value of 8 will only compute the 8 most significant byte of the result, with the remaining 3 bytes as 0. The maximum value for this parameter is VECT_SQRT_S32_MAX_DEPTH (31). The time cost of this operation is approximately proportional to the number of bits computed.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$a_k \leftarrow \sqrt{b'_k}$$

    for $k \in 0 \ldots (length - 1)$

    where $sqrt()$ computes the first $depth$ bits of the square root.

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = (b\_exp + b\_shr - 30)/2$.

Note that because exponents must be integers, that means $b\_exp + b\_shr$ **must be even**.

The function *vect_s32_sqrt_prepare()* can be used to obtain values for a_exp and b_shr based on the input exponent b_exp and headroom b_hr.

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- **b_shr** – **[in]** Right-shift appled to $\bar{b}$
- **depth** – **[in]** Number of bits of each output value to compute

Throws ET_LOAD_STORE

    Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

Returns

    Headroom of output vector $\bar{a}$

headroom_t **vect_s32_sub**(int32_t a[], const int32_t b[], const int32_t c[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Subtract one 32-bit vector from another.

a[], b[] and c[] represent the 32-bit mantissa vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[] or c[].

`length` is the number of elements in each of the vectors.

`b_shr` and `c_shr` are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k = sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k = sat_{32}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$a_k \leftarrow sat_{32}(b'_k - c'_k)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$.

In this case, b_shr and c_shr **must** be chosen so that $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$. Adding or subtracting mantissas only makes sense if they are associated with the same exponent.

The function *vect_s32_sub_prepare()* can be used to obtain values for a_exp, b_shr and $* c\_shr$ based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

See also:

vect_s32_sub_prepare

Parameters

- **a** − **[out]** Output vector $\bar{a}$
- **b** − **[in]** Input vector $\bar{b}$
- **c** − **[in]** Input vector $\bar{c}$
- **length** − **[in]** Number of elements in vectors $\bar{a}, \bar{b}$ and $\bar{c}$
- **b_shr** − **[in]** Right-shift appled to $\bar{b}$
- **c_shr** − **[in]** Right-shift appled to $\bar{c}$

Throws ET_LOAD_STORE
Raised if `a`, `b` or `c` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of output vector $\bar{a}$

int64_t **vect_s32_sum**(const int32_t b[], const unsigned length)

Sum the elements of a 32-bit vector.

`b[]` represents the 32-bit mantissa vector $\bar{b}$. `b[]` must begin at a word-aligned address.

`length` is the number of elements in $\bar{b}$.

Operation Performed

$$a \leftarrow \sum_{k=0}^{length-1} b_k$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the returned value a is the 64-bit mantissa of floating-point value $a \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Additional Details

Internally, each element accumulates into one of eight 40-bit accumulators (which are all used simultaneously) which apply symmetric 40-bit saturation logic (with bounds $\approx 2^{39}$) with each value added. The saturating arithmetic employed is *not associative* and no indication is given if saturation occurs at an intermediate step. To avoid the possibility of saturation errors, `length` should be no greater than 2^{11+b_hr}, where b_hr is the headroom of $\bar{b}$.

If the caller's mantissa vector is longer than that, the full result can be found by calling this function multiple times for partial results on sub-sequences of the input, and adding the results in user code.

In many situations the caller may have *a priori* knowledge that saturation is impossible (or very nearly so), in which case this guideline may be disregarded. However, such situations are application-specific and are well beyond the scope of this documentation, and as such are left to the user's discretion.

Parameters

- **b** – **[in]** Input vector $\bar{b}$

- **length** – **[in]** Number of elements in vector $\bar{b}$

Throws ET_LOAD_STORE
Raised if b is not word-aligned (See *Note: Vector Alignment*)

Returns
64-bit mantissa of the sum, a.

void **vect_s32_zip**(*complex_s32_t* a[], const int32_t b[], const int32_t c[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Interleave the elements of two vectors into a single vector.

Elements of 32-bit input vectors $\bar{b}$ and $\bar{c}$ are interleaved into 32-bit output vector $\bar{a}$. Each element of $\bar{b}$ has a right-shift of b_shr applied, and each element of $\bar{c}$ has a right-shift of c_shr applied.

Alternatively (and equivalently), this function can be conceived of as taking two real vectors $\bar{b}$ and $\bar{c}$ and forming a new complex vector $\bar{a}$ where $\bar{a} = \bar{b} + i \cdot \bar{c}$.

If vectors $\bar{b}$ and $\bar{c}$ each have N elements, then the resulting $\bar{a}$ will have either $2N$ int32_t elements or (equivalently) N `complex_s32_t` elements (and must have space for such).

Each element b_k of $\bar{b}$ will end up as end up as element a_{2k} of $\bar{a}$ (with the bit-shift applied). Each element c_k will end up as element a_{2k+1} of $\bar{a}$.

a[] is the output vector $\bar{a}$.

b[] and c[] are the input vectors $\bar{b}$ and $\bar{c}$ respectively.

a, b and c must each begin at a double word-aligned (8 byte) address. (see DWORD_ALIGNED).

length is the number N of int32_t elements in $\bar{b}$ and $\bar{c}$.

b_shr is the signed arithmetic right-shift applied to elements of $\bar{b}$.

c_shr is the signed arithmetic right-shift applied to elements of $\bar{c}$.

Operation Performed

$$Rea_k \leftarrow sat_{32}(b_k \cdot 2^{-b\_shr}$$
$$Ima_k \leftarrow sat_{32}(c_k \cdot 2^{-c\_shr}$$
$$\text{for } k \in 0 \ldots (N-1)$$

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **c** – **[in]** Input vector $\bar{c}$
- **length** – **[in]** Number of elements N in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **b_shr** – **[in]** Signed arithmetic right-shift applied to elements of $\bar{b}$
- **c_shr** – **[in]** Signed arithmetic right-shift applied to elements of $\bar{c}$

Throws ET_LOAD_STORE
Raised if a, b or c is not double word-aligned (See *Note: Vector Alignment*)

void **vect_s32_unzip**(int32_t a[], int32_t b[], const *complex_s32_t* c[], const unsigned length)

Deinterleave the real and imaginary parts of a complex 32-bit vector into two separate vectors.

Complex 32-bit input vector $\bar{c}$ has its real and imaginary parts (which correspond to the even and odd-indexed elements, if reinterpreted as an int32_t array) split apart to create real 32-bit output vectors $\bar{a}$ and $\bar{b}$, such that $\bar{a} = Re\bar{c}$ and $\bar{b} = Im\bar{c}$.

a[] and b[] are the real output vectors $\bar{a}$ and $\bar{b}$ which receive the real and imaginary parts respectively of $\bar{c}$. a and b must each begin at a word-aligned address.

c[] is the complex input vector $\bar{c}$. c must begin at a double word-aligned address.

length is the number N of int32_t elements in $\bar{a}$ and $\bar{b}$ and the number of *complex_s32_t* in $\bar{c}$.

Operation Performed

$$a_k = Re\{c_k\}$$
$$b_k = Im\{c_k\}$$
$$\text{for } k \in 0 \ldots (N-1)$$

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[out]** Output vector $\bar{b}$
- **c** – **[in]** Input vector $\bar{c}$
- **length** – **[in]** The number of elements N in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$

Throws ET_LOAD_STORE
Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

Throws ET_LOAD_STORE
Raised if c is not double word-aligned (See *Note: Vector Alignment*)

headroom_t **vect_s32_convolve_valid**(int32_t y[], const int32_t x[], const int32_t b_q30[], const
unsigned x_length, const unsigned b_length)

Convolve a 32-bit vector with a short kernel.

32-bit input vector $\bar{x}$ is convolved with a short fixed-point kernel $\bar{b}$ to produce 32-bit output vector $\bar{y}$. In other words, this function applies the Kth-order FIR filter with coefficients given by $\bar{b}$ to the input signal $\bar{x}$. The convolution is "valid" in the sense that no output elements are emitted where the filter taps extend beyond the bounds of the input vector, resulting in an output vector $\bar{y}$ with fewer elements.

The maximum filter order K supported by this function is 7.

y[] is the output vector $\bar{y}$. If input $\bar{x}$ has N elements, and the filter has K elements, then $\bar{y}$ has $N - 2P$ elements, where $P = \lfloor K/2 \rfloor$.

x[] is the input vector $\bar{x}$ with length N.

b_q30[] is the vector $\bar{b}$ of filter coefficients. The coefficients of $\bar{b}$ are encoded in a Q2.30 fixed-point format. The effective value of the ith coefficient is then $b_i \cdot 2^{-30}$.

x_length is the length N of $\bar{x}$ in elements.

b_length is the length K of $\bar{b}$ in elements (i.e. the number of filter taps). b_length must be one of $\{1, 3, 5, 7\}$.

Operation Performed

$$y_k \leftarrow \sum_{l=0}^{K-1} (x_{(k+l)} \cdot b_l \cdot 2^{-30})$$
$$\text{for } k \in 0 \ldots (N - 2P)$$
$$\text{where } P = \lfloor K/2 \rfloor$$

Additional Details

To avoid the possibility of saturating any output elements, $\bar{b}$ may be constrained such that $\sum_{i=0}^{K-1} |b_i| \leq 2^{30}$.

This operation can be applied safely in-place on x[].

Parameters

- **y** − **[out]** Output vector $\bar{y}$

- **x** − **[in]** Input vector $\bar{x}$

- **b_q30** − **[in]** Filter coefficient vector $\bar{b}$

- **x_length** − **[in]** The number of elements N in vector $\bar{x}$

- **b_length** − **[in]** The number of elements K in $\bar{b}$

Throws ET_LOAD_STORE
Raised if x or y or b_q30 is not word-aligned (See *Note: Vector Alignment*)

headroom_t **vect_s32_convolve_same**(int32_t y[], const int32_t x[], const int32_t b_q30[], const
unsigned x_length, const unsigned b_length, const
pad_mode_e padding_mode)

Convolve a 32-bit vector with a short kernel.

32-bit input vector $\bar{x}$ is convolved with a short fixed-point kernel $\bar{b}$ to produce 32-bit output vector $\bar{y}$. In other words, this function applies the Kth-order FIR filter with coefficients given by $\bar{b}$ to the input signal $\bar{x}$. The convolution mode is "same" in that the input vector is effectively padded such that the input and output vectors are the same length. The padding behavior is one of those given by *pad_mode_e*.

The maximum filter order K supported by this function is 7.

y[] and x[] are the output and input vectors $\bar{y}$ and $\bar{x}$ respectively.

b_q30[] is the vector $\bar{b}$ of filter coefficients. The coefficients of $\bar{b}$ are encoded in a Q2.30 fixed-point format. The effective value of the ith coefficient is then $b_i \cdot 2^{-30}$.

x_length is the length N of $\bar{x}$ and $\bar{y}$ in elements.

b_length is the length K of $\bar{b}$ in elements (i.e. the number of filter taps). b_length must be one of $\{1, 3, 5, 7\}$.

padding_mode is one of the values from the *pad_mode_e* enumeration. The padding mode indicates the filter input values for filter taps that have extended beyond the bounds of the input vector $\bar{x}$. See *pad_mode_e* for a list of supported padding modes and associated behaviors.

Operation Performed

$$\tilde{x}_i = \begin{cases} \text{determined by padding mode} & i < 0 \\ \text{determined by padding mode} & i \geq N \\ x_i & otherwise \end{cases}$$

$$y_k \leftarrow \sum_{l=0}^{K-1} (\tilde{x}_{(k+l-P)} \cdot b_l \cdot 2^{-30})$$

$$\text{for } k \in 0 \dots (N - 2P)$$

$$\text{where } P = \lfloor K/2 \rfloor$$

Additional Details

To avoid the possibility of saturating any output elements, $\bar{b}$ may be constrained such that $\sum_{i=0}^{K-1} |b_i| \leq 2^{30}$.

Note: Unlike *vect_s32_convolve_valid()*, this operation *cannot* be performed safely in-place on x[]

Parameters

- **y** – **[out]** Output vector $\bar{y}$
- **x** – **[in]** Input vector $\bar{x}$
- **b_q30** – **[in]** Filter coefficient vector $\bar{b}$
- **x_length** – **[in]** The number of elements N in vector $\bar{x}$
- **b_length** – **[in]** The number of elements K in $\bar{b}$
- **padding_mode** – **[in]** The padding mode to be applied at signal boundaries

Throws ET_LOAD_STORE
Raised if x or y or b_q30 is not word-aligned (See *Note: Vector Alignment*)

void **vect_s32_merge_accs**(int32_t a[], const *split_acc_s32_t* b[], const unsigned length)

Merge a vector of split 32-bit accumulators into a vector of int32_t's.

Convert a vector of *split_acc_s32_t* into a vector of `int32_t`. This is useful when a function (e.g. `mat_mul_s8_x_s8_yield_s32`) outputs a vector of accumulators in the XS3 VPU's native split 32-bit format, which has the upper half of each accumulator in the first 32 bytes and the lower half in the following 32 bytes.

This function is most efficient (in terms of cycles/accumulator) when `length` is a multiple of

a. In any case, `length` will be rounded up such that a multiple of 16 accumulators will always be merged.

This function can safely merge accumulators in-place.

Parameters

- **a** – **[out]** Output vector of int32_t
- **b** – **[in]** Input vector of *split_acc_s32_t*
- **length** – **[in]** Number of accumulators to merge

Throws ET_LOAD_STORE

Raised if `b` or `a` is not word-aligned (See *Note: Vector Alignment*)

void **vect_s32_split_accs**(*split_acc_s32_t* a[], const int32_t b[], const unsigned length)

Split a vector of `int32_t`'s into a vector of *split_acc_s32_t*.

Convert a vector of `int32_t` into a vector of *split_acc_s32_t*, the native format for the XS3 VPU's 32-bit accumulators. This is useful when a function (e.g. `mat_mul_s8_x_s8_yield_s32`) takes in a vector of accumulators in that native format.

This function is most efficient (in terms of cycles/accumulator) when `length` is a multiple of

a. In any case, `length` will be rounded up such that a multiple of 16 accumulators will always be split.

This function can safely split accumulators in-place.

Parameters

- **a** – **[out]** Output vector of *split_acc_s32_t*
- **b** – **[in]** Input vector of int32_t
- **length** – **[in]** Number of accumulators to split

Throws ET_LOAD_STORE

Raised if `b` or `a` is not word-aligned (See *Note: Vector Alignment*)

void **vect_split_acc_s32_shr**(*split_acc_s32_t* a[], const unsigned length, const *right_shift_t* shr)

Apply a right-shift to the elements of a 32-bit split accumulator vector.

This function may be used in conjunction with *chunk_s16_accumulate()* or *bfp_s16_accumulate()* to avoid saturation of accumulators.

This function updates $\bar{a}$ in-place.

Parameters

- **a** – **[inout]** Accumulator vector $\bar{a}$
- **length** – **[in]** Number of elements of $\bar{a}$
- **shr** – **[in]** Number of bits to right-shift the elements of $\bar{a}$

Throws ET_LOAD_STORE

Raised if `a` is not double-word-aligned (See *Note: Vector Alignment*)

void **vect_q30_power_series**(int32_t a[], const *q2_30* b[], const int32_t c[], const unsigned term_count, const unsigned length)

Compute a power series sum on a vector of Q2.30 values.

This function is used to compute a power series summation on a vector $\bar{b}$. $\bar{b}$ contains Q2.30 values. $\bar{c}$ is a vector containing coefficients to be multiplied by powers of $\bar{b}$, and may have any associated exponent. The output is vector $\bar{a}$ and has the same exponent as $\bar{c}$.

c[] is an array with shape (`term_count`, `VPU_INT32_EPV`), where the second axis contains the same value replicated across all `VPU_INT32_EPV` elements. That is, `c[k][i] = c[k][j]` for i and j in 0..(`VPU_INT32_EPV-1`). This is for performance reasons. (For the purpose of this explanation, $\bar{c}$ is considered to be single-dimensional, without redundancy.)

Operation Performed

$$b_{k,0} = 2^{30}$$

$$b_{k,i} = round\left(\frac{b_{k,i-1} \cdot b_k}{2^{30}}\right)$$

$$\text{for } i \in 1..(N-1)$$

$$a_k \leftarrow \sum_{i=0}^{N-1} round\left(\frac{b_{k,i} \cdot c_i}{2^{30}}\right)$$

$$\text{for } k \in 0..\text{length} - 1$$

Parameters

- **a** − **[out]** Output vector $\bar{a}$

- **b** − **[in]** Input vector $\bar{b}$

- **c** − **[in]** Coefficient vector $\bar{c}$

- **term_count** − **[in]** Number of power series terms, N

- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

void **vect_float_s32_log_base**(*q8_24* a[], const *float_s32_t* b[], const *q2_30* inv_ln_base_q30, const unsigned length)

Compute the logarithm (in the specified base) of a vector of *float_s32_t*.

This function computes the logarithm of a vector $\bar{b}$ of *float_s32_t* values. The base of the computed logarithm is given by parameter `inv_ln_base_q30`. The result is written to output $\bar{a}$, a vector of Q8.24 values.

If the desired base is D, then `inv_ln_base_q30`, represented here by R, should be $\text{Q30}\left(\frac{1}{ln(D)}\right)$. That is: the inverse of the natural logarithm of the desired base, expressed as a Q2.30 value. Typically the desired base is known at compile time, so this value will usually be a precomputed constant.

The resulting a_k for $b_k \leq 0$ is undefined.

Operation Performed

$$a_k \leftarrow log_D(b_k)$$

$$\text{for } k \in 0..\text{length} - 1$$

Parameters

- **a** − **[out]** Output Q8.24 vector $\bar{a}$

- **b** − **[in]** Input vector $\bar{b}$

- **inv_ln_base_q30** − **[in]** Coefficient R converting from natural log to desired base D

- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE
Raised if b or a is not double word-aligned (See *Note: Vector Alignment*)

void **vect_float_s32_log**(*q8_24* a[], const *float_s32_t* b[], const unsigned length)

Compute the natural logarithm of a vector of *float_s32_t*.

This function computes the natural logarithm of a vector $\bar{b}$ of *float_s32_t* values. The result is written to output $\bar{a}$, a vector of Q8.24 values.

The resulting a_k for $b_k \leq 0$ is undefined.

Operation Performed

$$a_k \leftarrow ln\,(b_k)$$
$$\text{for } k \in 0..\texttt{length} - 1$$

Parameters

- **a** − **[out]** Output Q8.24 vector $\bar{a}$

- **b** − **[in]** Input vector $\bar{b}$

- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE
Raised if b or a is not double word-aligned (See *Note: Vector Alignment*)

void **vect_float_s32_log2**(*q8_24* a[], const *float_s32_t* b[], const unsigned length)

Compute the base 2 logarithm of a vector of *float_s32_t*.

This function computes the base 2 logarithm of a vector $\bar{b}$ of *float_s32_t* values. The result is written to output $\bar{a}$, a vector of Q8.24 values.

The resulting a_k for $b_k \leq 0$ is undefined.

Operation Performed

$$a_k \leftarrow log_2\,(b_k)$$
$$\text{for } k \in 0..\texttt{length} - 1$$

Parameters

- **a** − **[out]** Output Q8.24 vector $\bar{a}$

- **b** − **[in]** Input vector $\bar{b}$

- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE
Raised if b or a is not double word-aligned (See *Note: Vector Alignment*)

void **vect_float_s32_log10**(*q8_24* a[], const *float_s32_t* b[], const unsigned length)

Compute the base 10 logarithm of a vector of *float_s32_t*.

This function computes the base 10 logarithm of a vector $\bar{b}$ of *float_s32_t* values. The result is written to output $\bar{a}$, a vector of Q8.24 values.

The resulting a_k for $b_k \leq 0$ is undefined.

Operation Performed

$$a_k \leftarrow log_{10}(b_k)$$
$$\text{for } k \in 0..\texttt{length} - 1$$

Parameters

- **a** – **[out]** Output Q8.24 vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE
Raised if b or a is not double word-aligned (See *Note: Vector Alignment*)

void **vect_s32_log_base**(*q8_24* a[], const int32_t b[], const *exponent_t* b_exp, const *q2_30* inv_ln_base_q30, const unsigned length)

Compute the logarithm (in the specified base) of a block floating-point vector.

This function computes the logarithm of the block floating-point vector $\bar{b} \cdot 2^{b\_exp}$. The base of the computed logarithm is given by parameter `inv_ln_base_q30`. The result is written to output $\bar{a}$, a vector of Q8.24 values.

If the desired base is D, then `inv_ln_base_q30`, represented here by R, should be Q30 $\left(\frac{1}{ln(D)}\right)$. That is: the inverse of the natural logarithm of the desired base, expressed as a Q2.30 value. Typically the desired base is known at compile time, so this value will usually be a precomputed constant.

The resulting a_k for $b_k \leq 0$ is undefined.

Operation Performed

$$a_k \leftarrow log_D\left(b_k \cdot 2^{b\_exp}\right)$$
$$\text{for } k \in 0..\texttt{length} - 1$$

Parameters

- **a** – **[out]** Output Q8.24 vector $\bar{a}$
- **b** – **[in]** Input mantissa vector $\bar{b}$
- **b_exp** – **[in]** Exponent associated with $\bar{b}$
- **inv_ln_base_q30** – **[in]** Coefficient R converting from natural log to desired base D
- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE
Raised if b or a is not double word-aligned (See *Note: Vector Alignment*)

void **vect_s32_log**(*q8_24* a[], const int32_t b[], const *exponent_t* b_exp, const unsigned length)

Compute the natural logarithm of a block floating-point vector.

This function computes the natural logarithm of the block floating-point vector $\bar{b} \cdot 2^{b\_exp}$. The result is written to output $\bar{a}$, a vector of Q8.24 values.

The resulting a_k for $b_k \leq 0$ is undefined.

Operation Performed

$$a_k \leftarrow ln\left(b_k \cdot 2^{b\_exp}\right)$$
$$\text{for } k \in 0..\texttt{length} - 1$$

Parameters

- **a** – **[out]** Output Q8.24 vector $\bar{a}$
- **b** – **[in]** Input mantissa vector $\bar{b}$
- **b_exp** – **[in]** Exponent associated with $\bar{b}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE
 Raised if b or a is not double word-aligned (See *Note: Vector Alignment*)

void **vect_s32_log2**(*q8_24* a[], const int32_t b[], const *exponent_t* b_exp, const unsigned length)

Compute the base 2 logarithm of a block floating-point vector.

This function computes the base 2 logarithm of the block floating-point vector $\bar{b} \cdot 2^{b\_exp}$. The result is written to output $\bar{a}$, a vector of Q8.24 values.

The resulting a_k for $b_k \leq 0$ is undefined.

Operation Performed

$$a_k \leftarrow log_2\left(b_k \cdot 2^{b\_exp}\right)$$
$$\text{for } k \in 0..\texttt{length} - 1$$

Parameters

- **a** – **[out]** Output Q8.24 vector $\bar{a}$
- **b** – **[in]** Input mantissa vector $\bar{b}$
- **b_exp** – **[in]** Exponent associated with $\bar{b}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE
 Raised if b or a is not double word-aligned (See *Note: Vector Alignment*)

void **vect_s32_log10**(*q8_24* a[], const int32_t b[], const *exponent_t* b_exp, const unsigned length)

Compute the base 10 logarithm of a block floating-point vector.

This function computes the base 10 logarithm of the block floating-point vector $\bar{b} \cdot 2^{b\_exp}$. The result is written to output $\bar{a}$, a vector of Q8.24 values.

The resulting a_k for $b_k \leq 0$ is undefined.

Operation Performed

$$a_k \leftarrow log_{10}\left(b_k \cdot 2^{b\_exp}\right)$$
$$\text{for } k \in 0..\texttt{length} - 1$$

Parameters

- **a** – **[out]** Output Q8.24 vector $\bar{a}$
- **b** – **[in]** Input mantissa vector $\bar{b}$

- **b_exp** – **[in]** Exponent associated with $\bar{b}$

- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE

Raised if b or a is not double word-aligned (See *Note: Vector Alignment*)

void **vect_q30_exp_small**(*q2_30* a[], const *q2_30* b[], const unsigned length)

Compute e^x for Q2.30 value near 0.

This function computes $e^{b_k \cdot 2^{-30}}$ for each b_k in input vector $\bar{b}$. The results are placed in output vector $\bar{a}$ as Q2.30 values.

This function is meant to compute e^x for values of x in the interval $[-0.5, 0.5]$. The error grows quickly outside of this range.

Operation Performed

$$a_k \leftarrow \frac{e^{b_k \cdot 2^{-30}}}{2^{30}}$$
$$\text{for } k \in 0..(\texttt{length} - 1)$$

Parameters

- **a** – **[out]** Output vector $\bar{a}$

- **b** – **[in]** Input vector $\bar{b}$

- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

void **vect_s32_to_vect_s16**(int16_t a[], const int32_t b[], const unsigned length, const *right_shift_t* b_shr)

Convert a 32-bit vector to a 16-bit vector.

This function converts a 32-bit mantissa vector $\bar{b}$ into a 16-bit mantissa vector $\bar{a}$. Conceptually, the output BFP vector $\bar{a} \cdot 2^{a\text{-}exp}$ represents the same values as the input BFP vector $\bar{b} \cdot 2^{b\text{-}exp}$, only with a reduced bit-depth.

In most cases b_shr should be $16 - b\_hr$, where b_hr is the headroom of the 32-bit input mantissa vector $\bar{b}$.

The output exponent a_exp will be given by

$a\_exp = b\_exp + b\_shr$

Parameter Details

a[] represents the 16-bit output mantissa vector $\bar{a}$.

b[] represents the 32-bit input mantissa vector $\bar{b}$.

a[] and b[] must each begin at a word-aligned address.

length is the number of elements in each of the vectors.

b_shr is the signed arithmetic right-shift applied to elements of $\bar{b}$.

Operation Performed

$$a_k \leftarrow sat_{16}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the 32-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the 16-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + b\_shr$.

See also:

vect_s16_to_vect_s32

Parameters

- **a** − **[out]** Output vector $\bar{a}$
- **b** − **[in]** Input vector $\bar{b}$
- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- **b_shr** − **[in]** Right-shift appled to $\bar{b}$

Throws ET_LOAD_STORE
Raised if `a` or `b` is not word-aligned (See *Note: Vector Alignment*)

4.7.5   32-Bit IEEE 754 Float API

group **vect_f32_api**

Functions

complex_float_t ***fft_f32_forward**(float x[], const unsigned fft_length)
Perform forward FFT on a vector of IEEE754 floats.

This function takes real input vector $\bar{x}$ and performs a forward FFT on the signal in-place to get output vector $\bar{X} = FFT\bar{x}$. This implementation is accelerated by converting the IEEE754 float vector into a block floating-point representation to compute the FFT. The resulting BFP spectrum is then converted back to IEEE754 single-precision floats. The operation is performed in-place on `x[]`.

See *bfp_fft_forward_mono()* for the details of the FFT.

Whereas the input `x[]` is an array of `fft_length` `float` elements, the output (placed in `x[]`) is an array of `fft_length/2` *complex_float_t* elements, so the input should be cast after calling this.

```
const unsigned FFT_N = 512
float time_series[FFT_N] = { ... };
fft_f32_forward(time_series, FFT_N);
complex_float_t* freq_spectrum = (complex_float_t*) &time_series[0];
const unsigned FREQ_BINS = FFT_N/2;
// e.g.   freq_spectrum[FREQ_BINS-1].re
```

`x[]` must begin at a double-word-aligned address.

Operation Performed

$$\bar{X} \leftarrow FFT\bar{x}$$

> **Parameters**
>
> - **x** – **[inout]** Input vector $\bar{x}$
> - `fft_length` – **[in]** The length of $\bar{x}$
>
> **Throws ET_LOAD_STORE**
> Raised if `x` is not double-word-aligned (See *Note: Vector Alignment*)
>
> **Returns**
> Pointer to frequency-domain spectrum (i.e. `((complex_float_t*) &x[0])`)

float *`fft_f32_inverse`(*complex_float_t* X[], const unsigned fft_length)

Perform inverse FFT on a vector of *complex_float_t*.

This function takes complex input vector $\bar{X}$ and performs an inverse real FFT on the spectrum in-place to get output vector $\bar{x} = IFFT\bar{X}$. This implementation is accelerated by converting the IEEE754 float vector into a block floating-point representation to compute the IFFT. The resulting BFP signal is then converted back to IEEE754 single-precision floats. The operation is performed in-place on `X[]`.

See *bfp_fft_inverse_mono()* for the details of the IFFT.

Input `X[]` is an array of `fft_length/2` *complex_float_t* elements. The output (placed in `X[]`) is an array of `fft_length` float elements.

```
const unsigned FFT_N = 512
complex_float_t freq_spectrum[FFT_N/2] = { ... };
fft_f32_inverse(freq_spectrum, FFT_N);
float* time_series = (float*) &freq_spectrum[0];
```

`X[]` must begin at a double-word-aligned address.

> **Parameters**
>
> - **x** – **[inout]** Input vector $\bar{X}$
> - `fft_length` – **[in]** The FFT length. Twice the element count of $\bar{X}$.
>
> **Throws ET_LOAD_STORE**
> Raised if `X` is not double-word-aligned (See *Note: Vector Alignment*)
>
> **Returns**
> Pointer to time-domain signal (i.e. `((float*) &X[0])`)

exponent_t **vect_f32_max_exponent**(const float b[], const unsigned length)

Get the maximum (32-bit BFP) exponent from a vector of IEEE754 floats.

This function is used to determine the BFP exponent to use when converting a vector of IEEE754 single-precision floats into a 32-bit BFP vector.

The exponent returned, if used with *vect_f32_to_vect_s32()*, is the one which will result in no headroom in the BFP vector — that is, the *minimum* permissible exponent for the BFP vector. The minimum permissible exponent is derived from the *maximum* exponent found in the `float` elements themselves.

More specifically, the `FSEXP` instruction is used on each element to determine its exponent. The value returned is the maximum exponent given by the `FSEXP` instruction plus `30`.

`b[]` must begin at a double-word-aligned address.

Note: If required, when converting to a 32-bit BFP vector, additional headroom can be included by adding the amount of required headroom to the exponent returned by this function.

Parameters

- **b** − **[in]** Input vector of IEEE754 single-precision floats $\bar{b}$

- **length** − **[in]** Number of elements in $\bar{b}$

Throws ET_LOAD_STORE
    Raised if b is not double-word-aligned (See *Note: Vector Alignment*)

Throws ET_ARITHMETIC
    Raised if Any element of b is infinite or not-a-number.

Returns
    Exponent used for converting to 32-bit BFP vector.

void **vect_f32_to_vect_s32**(int32_t a[], const float b[], const unsigned length, const *exponent_t* a_exp)

Convert a vector of IEEE754 single-precision floats into a 32-bit BFP vector.

This function converts a vector of IEEE754 single-precision floats $\bar{b}$ into the mantissa vector $\bar{a}$ of a 32-bit BFP vector, given BFP vector exponent a_exp. Conceptually, the elements of output vector $\bar{a} \cdot 2^{a\_exp}$ represent the same values as those of the input vector.

Because the output exponent a_exp is shared by all elements of the output vector, even though the output vector has 32-bit mantissas, precision may be lost on some elements if the exponents of the input elements b_k span a wide range.

The function *vect_f32_max_exponent()* can be used to determine the value for a_exp which minimizes headroom of the output vector.

Operation Performed

$$a_k \leftarrow round(\frac{b_k}{2^{b\_exp}})$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Parameter Details

a[ ] represents the 32-bit output mantissa vector $\bar{a}$.

b[ ] represents the IEEE754 float input vector $\bar{b}$.

a[ ] and b[ ] must each begin at a double-word-aligned address.

b[ ] can be safely updated in-place.

length is the number of elements in each of the vectors.

a_exp is the exponent associated with the output vector $\bar{a}$.

See also:

vect_f32_max_exponent

See also:

vect_s32_to_vect_f32

> **Parameters**
>
> - **a** – **[out]** Output vector $\bar{a}$
> - **b** – **[in]** Input vector $\bar{b}$
> - **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
> - **a_exp** – **[in]** Exponent a_exp of output vector $\bar{a}$
>
> **Throws ET_LOAD_STORE**
> Raised if `a` or `b` is not double-word-aligned (See *Note: Vector Alignment*)
>
> **Throws ET_ARITHMETIC**
> Raised if Any element of `b` is infinite or not-a-number.

float **vect_f32_dot**(const float b[], const float c[], const unsigned length)

Compute the inner product of two IEEE754 float vectors.

This function takes two vectors of IEEE754 single-precision floats and computes their inner product — the sum of the elementwise products. The `FMACC` instruction is used, granting full precision in the addition.

The inner product a is returned.

Operation Performed

$$a \leftarrow \sum_{k=0}^{length-1} (b_k \cdot c_k)$$

> **Parameters**
>
> - **b** – **[in]** Input vector $\bar{b}$
> - **c** – **[in]** Input vector $\bar{c}$
> - **length** – **[in]** Number of elements in vectors $\bar{b}$ and $\bar{c}$
>
> **Returns**
> The inner product

void **vect_f32_add**(float a[], const float b[], const float c[], const unsigned length)

Adds together two IEEE754 float vectors.

This function takes two vectors of IEEE754 single-precision floats and computes the element-wise sum of the two vectors.

`a[]` is the output vector $\bar{a}$ into which results are placed.

`b[]` and `c[]` are the input vectors $\bar{b}$ and $\bar{c}$ respectively.

`a`, `b` and `c` each must begin at a double-word-aligned address.

This operation can be performed safely in-place on `b[]` or `c[]`.

Operation Performed

$$a_k \leftarrow b_k + c_k$$
$$\text{for } k \in 0 \dots (length - 1)$$

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **c** – **[in]** Input vector $\bar{c}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$

Throws ET_LOAD_STORE
Raised if a, b or c is not double-word-aligned (See *Note: Vector Alignment*)

void **vect_complex_f32_add**(*complex_float_t* a[], const *complex_float_t* b[], const *complex_float_t* c[], const unsigned length)

Adds together two complex IEEE754 float vectors.

This function takes two vectors $\bar{b}$ and $\bar{c}$ of complex IEEE754 single-precision floats and computes the element-wise sum of the two vectors.

a[] is the output vector $\bar{a}$ into which results are placed.

b[] and c[] are the complex input vectors $\bar{b}$ and $\bar{c}$ respectively.

a, b and c each must begin at a double-word-aligned address.

This operation can be performed safely in-place on b[] or c[].

Operation Performed

$$a_k \leftarrow b_k + c_k$$
$$\text{for } k \in 0 \dots (length - 1)$$

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **c** – **[in]** Input vector $\bar{c}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$

Throws ET_LOAD_STORE
Raised if a, b or c is not double-word-aligned (See *Note: Vector Alignment*)

void **vect_complex_f32_mul**(*complex_float_t* a[], const *complex_float_t* b[], const *complex_float_t* c[], const unsigned length)

Multiplies together two complex IEEE754 float vectors.

This function takes two complex float vectors $\bar{b}$ and $\bar{c}$ as inputs. Each output element a_k is computed as b_k multiplied by c_k (using complex multiplication).

a[] is the output vector $\bar{a}$ into which results are placed.

b[] and c[] are the complex input vectors $\bar{b}$ and $\bar{c}$ respectively.

a, b and c each must begin at a double-word-aligned address.

This operation can be performed safely in-place on b[] or c[].

Operation Performed

$$a_k \leftarrow b_k \cdot c_k$$
$$\text{for } k \in 0 \dots (length - 1)$$

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **c** – **[in]** Input vector $\bar{c}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$

Throws ET_LOAD_STORE
Raised if a, b or c is not double-word-aligned (See *Note: Vector Alignment*)

void **vect_complex_f32_conj_mul**(*complex_float_t* a[], const *complex_float_t* b[], const *complex_float_t* c[], const unsigned length)

Conjugate multiplies together two complex IEEE754 float vectors.

This function takes two complex float vectors $\bar{b}$ and $\bar{c}$ as inputs. Each output element a_k is computed as b_k multiplied by the complex conjugate of c_k (using complex multiplication).

a[] is the output vector $\bar{a}$ into which results are placed.

b[] and c[] are the complex input vectors $\bar{b}$ and $\bar{c}$ respectively.

a, b and c each must begin at a double-word-aligned address.

This operation can be performed safely in-place on b[] or c[].

Operation Performed

$$a_k \leftarrow b_k \cdot (c_k^*)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Parameters

- **a** – **[out]** Output vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **c** – **[in]** Input vector $\bar{c}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$

Throws ET_LOAD_STORE
Raised if a, b or c is not double-word-aligned (See *Note: Vector Alignment*)

void **vect_complex_f32_macc**(*complex_float_t* a[], const *complex_float_t* b[], const *complex_float_t* c[], const unsigned length)

Adds the product of two complex IEEE754 float vectors to a third float vector.

This function takes three complex float vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$ as inputs. Each output element a_k is computed as input a_k plus b_k multiplied by c_k.

a[] is accumulator vector $\bar{a}$, serving as both input and output.

b[] and c[] are the complex input vectors $\bar{b}$ and $\bar{c}$ respectively.

a, b and c each must begin at a double-word-aligned address.

Operation Performed

$$a_k \leftarrow a_k + b_k \cdot c_k$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Parameters

- **a** – **[inout]** Input/Output accumulator vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **c** – **[in]** Input vector $\bar{c}$
- **length** – **[in]** Number of elements in vectors $\bar{a}, \bar{b}$ and $\bar{c}$

Throws ET_LOAD_STORE
Raised if a, b or c is not double-word-aligned (See *Note: Vector Alignment*)

void **vect_complex_f32_conj_macc**(*complex_float_t* a[], const *complex_float_t* b[], const *complex_float_t* c[], const unsigned length)

Adds the product of two complex IEEE754 float vectors to a third float vector.

This function takes three complex float vectors $\bar{a}, \bar{b}$ and $\bar{c}$ as inputs. Each output element a_k is computed as input a_k plus b_k multiplied by the complex conjugate of c_k.

a[] is accumulator vector $\bar{a}$, serving as both input and output.

b[] and c[] are the complex input vectors $\bar{b}$ and $\bar{c}$ respectively.

a, b and c each must begin at a double-word-aligned address.

Operation Performed

$$a_k \leftarrow a_k + b_k \cdot (c_k^*)$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Parameters

- **a** – **[inout]** Input/Output accumulator vector $\bar{a}$
- **b** – **[in]** Input vector $\bar{b}$
- **c** – **[in]** Input vector $\bar{c}$
- **length** – **[in]** Number of elements in vectors $\bar{a}, \bar{b}$ and $\bar{c}$

Throws ET_LOAD_STORE
Raised if a, b or c is not double-word-aligned (See *Note: Vector Alignment*)

void **vect_s32_to_vect_f32**(float a[], const int32_t b[], const unsigned length, const *exponent_t* b_exp)

Convert a 32-bit BFP vector into a vector of IEEE754 single-precision floats.

This function converts a 32-bit mantissa vector and exponent $\bar{b} \cdot 2^{b\_exp}$ into a vector of 32-bit IEEE754 single-precision floating-point elements $\bar{a}$. Conceptually, the elements of output vector $\bar{a}$ represent the same values as those of the input vector.

Because IEEE754 single-precision floats hold fewer mantissa bits, this operation may result in a loss of precision for some elements.

Operation Performed

$$a_k \leftarrow b_k \cdot 2^{b\_exp}$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Parameter Details

a[] represents the output IEEE754 float vector $\bar{a}$.

b[] represents the 32-bit input mantissa vector $\bar{b}$.

a[] and b[] must each begin at a double-word-aligned address.

b[] can be safely updated in-place.

length is the number of elements in each of the vectors.

b_exp is the exponent associated with the input vector $\bar{b}$.

See also:

vect_f32_to_vect_s32

Parameters

- **a** − **[out]** Output vector $\bar{a}$
- **b** − **[in]** Input vector $\bar{b}$
- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- **b_exp** − **[in]** Exponent b_exp of input vector $\bar{b}$

Throws ET_LOAD_STORE
Raised if a or b is not double-word-aligned (See *Note: Vector Alignment*)

4.7.6 Complex 16-Bit Vector API

group **vect_complex_s16_api**

Functions

headroom_t **vect_complex_s16_add**(int16_t a_real[], int16_t a_imag[], const int16_t b_real[], const int16_t b_imag[], const int16_t c_real[], const int16_t c_imag[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Add one complex 16-bit vector to another.

a_real[] and a_imag[] together represent the complex 16-bit output mantissa vector $\bar{a}$. Each $Re\{a_k\}$ is a_real[k], and each $Im\{a_k\}$ is a_imag[k].

b_real[] and b_imag[] together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is b_real[k], and each $Im\{b_k\}$ is b_imag[k].

c_real[] and c_imag[] together represent the complex 16-bit input mantissa vector $\bar{c}$. Each $Re\{c_k\}$ is c_real[k], and each $Im\{c_k\}$ is c_imag[k].

Each of the input vectors must begin at a word-aligned address. This operation can be performed safely in-place on inputs b_real[], b_imag[], c_real[] and c_imag[].

length is the number of elements in each of the vectors.

b_shr and c_shr are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k \leftarrow sat_{16}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k \leftarrow sat_{16}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$Re\{a_k\} \leftarrow Re\{b'_k\} + Re\{c'_k\}$$
$$Im\{a_k\} \leftarrow Im\{b'_k\} + Im\{c'_k\}$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the complex 16-bit mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the complex 16-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$.

In this case, b_shr and c_shr **must** be chosen so that $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$. Adding or subtracting mantissas only makes sense if they are associated with the same exponent.

The function *vect_complex_s16_add_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

See also:

vect_complex_s16_add_prepare

Parameters

- `a_real` – **[out]** Real part of complex output vector $\bar{a}$
- `a_imag` – **[out]** Imaginary aprt of complex output vector $\bar{a}$
- `b_real` – **[in]** Real part of complex input vector $\bar{b}$
- `b_imag` – **[in]** Imaginary part of complex input vector $\bar{b}$
- `c_real` – **[in]** Real part of complex input vector $\bar{c}$
- `c_imag` – **[in]** Imaginary part of complex input vector $\bar{c}$
- `length` – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- `b_shr` – **[in]** Right-shift applied to $\bar{b}$
- `c_shr` – **[in]** Right-shift applied to $\bar{c}$

Throws ET_LOAD_STORE
Raised if `a_real`, `a_imag`, `b_real`, `b_imag`, `c_real` or `c_imag` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of output vector $\bar{a}$.

headroom_t **vect_complex_s16_add_scalar**(int16_t a_real[], int16_t a_imag[], const int16_t b_real[], const int16_t b_imag[], const *complex_s16_t* c, const unsigned length, const *right_shift_t* b_shr)

Add a scalar to a complex 16-bit vector.

`a[]` and `b[]` represent the complex 16-bit mantissa vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on `b[]`.

`c` is the complex scalar c to be added to each element of $\bar{b}$.

`length` is the number of elements in each of the vectors.

`b_shr` is the signed arithmetic right-shift applied to each element of $\bar{b}$.

Operation Performed

$$b'_k \leftarrow sat_{16}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$Re\{a_k\} \leftarrow Re\{b'_k\} + Re\{c\}$$
$$Im\{a_k\} \leftarrow Im\{b'_k\} + Im\{c\}$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If elements of $\bar{b}$ are the complex mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, and c is the mantissa of floating-point value $c \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$.

In this case, b_shr and c_shr **must** be chosen so that $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$. Adding or subtracting mantissas only makes sense if they are associated with the same exponent.

The function *vect_complex_s16_add_scalar_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Note that c_shr is an output of `vect_complex_s16_add_scalar_prepare()`, but is not a parameter to this function. The c_shr produced by `vect_complex_s16_add_scalar_prepare()` is to be applied by the user, and the result passed as input `c`.

See also:

vect_complex_s16_add_scalar_prepare

Parameters

- `a_real` – **[out]** Real part of complex output vector $\bar{a}$
- `a_imag` – **[out]** Imaginary aprt of complex output vector $\bar{a}$
- `b_real` – **[in]** Real part of complex input vector $\bar{b}$
- `b_imag` – **[in]** Imaginary part of complex input vector $\bar{b}$
- `c` – **[in]** Complex input scalar c
- `length` – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- `b_shr` – **[in]** Right-shift applied to $\bar{b}$

Throws ET_LOAD_STORE
Raised if `a` or `b` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of output vector $\bar{a}$.

headroom_t **vect_complex_s16_conj_mul**(int16_t a_real[], int16_t a_imag[], const int16_t b_real[], const int16_t b_imag[], const int16_t c_real[], const int16_t c_imag[], const unsigned length, const *right_shift_t* a_shr)

Multiply one complex 16-bit vector element-wise by the complex conjugate of another.

`a_real[]` and `a_imag[]` together represent the complex 16-bit output mantissa vector $\bar{a}$. Each $Re\{a_k\}$ is `a_real[k]`, and each $Im\{a_k\}$ is `a_imag[k]`.

`b_real[]` and `b_imag[]` together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is `b_real[k]`, and each $Im\{b_k\}$ is `b_imag[k]`.

`c_real[]` and `c_imag[]` together represent the complex 16-bit input mantissa vector $\bar{c}$. Each $Re\{c_k\}$ is `c_real[k]`, and each $Im\{c_k\}$ is `c_imag[k]`.

Each of the input vectors must begin at a word-aligned address. This operation can be performed safely in-place on inputs `b_real[]`, `b_imag[]`, `c_real[]` and `c_imag[]`.

`length` is the number of elements in each of the vectors.

`a_shr` is the unsigned arithmetic right-shift applied to the 32-bit accumulators holding the penultimate results.

Operation Performed

$$v_k = \leftarrow Re\{b_k\} \cdot Re\{c_k\} + Im\{b_k\} \cdot Im\{c_k\}$$
$$s_k = \leftarrow Im\{b_k\} \cdot Re\{c_k\} - Re\{b_k\} \cdot Im\{c_k\}$$
$$Re\{a_k\} \leftarrow round(sat_{16}(v_k \cdot 2^{-a\_shr}))$$
$$Im\{a_k\} \leftarrow round(sat_{16}(s_k \cdot 2^{-a\_shr}))$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 16-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$ and c is the complex 16-bit mantissa of floating-point value $c \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + c\_exp + a\_shr$.

The function *vect_complex_s16_mul_prepare()* can be used to obtain values for a_exp and a_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

See also:

vect_complex_s16_mul_prepare

Parameters

- `a_real` – **[out]** Real part of complex output vector $\bar{a}$
- `a_imag` – **[out]** Imaginary aprt of complex output vector $\bar{a}$
- `b_real` – **[in]** Real part of complex input vector $\bar{b}$
- `b_imag` – **[in]** Imaginary part of complex input vector $\bar{b}$
- `c_real` – **[in]** Real part of complex input vector $\bar{c}$
- `c_imag` – **[in]** Imaginary part of complex input vector $\bar{c}$
- `length` – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- `a_shr` – **[in]** Right-shift applied to 32-bit intermediate results.

Throws ET_LOAD_STORE

Raised if `a_real`, `a_imag`, `b_real`, `b_imag`, `c_real` or `c_imag` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$

headroom_t **vect_complex_s16_headroom**(const int16_t b_real[], const int16_t b_imag[], const unsigned length)

Calculate the headroom of a complex 16-bit array.

The headroom of an N-bit integer is the number of bits that the integer's value may be left-shifted without any information being lost. Equivalently, it is one less than the number of leading sign bits.

The headroom of a *complex_s16_t* struct is the minimum of the headroom of each of its 16-bit fields, re and im.

The headroom of a *complex_s16_t* array is the minimum of the headroom of each of its *complex_s16_t* elements.

This function efficiently traverses the elements of $\bar{x}$ to determine its headroom.

b_real[] and b_imag[] together represent the complex 16-bit input mantissa vector $\bar{b}$.

length is the number of elements in b_real[] and b_imag[].

Operation Performed

$$min\{HR_{16}\left(x_0\right), HR_{16}\left(x_1\right), ..., HR_{16}\left(x_{length-1}\right)\}$$

See also:

vect_s16_headroom, *vect_s32_headroom*, *vect_complex_s32_headroom*

Parameters

- **b_real** – **[in]** Real part of complex input vector $\bar{b}$
- **b_imag** – **[in]** Imaginary part of complex input vector $\bar{b}$
- **length** – **[in]** Number of elements in $\bar{x}$

Returns
Headroom of vector $\bar{x}$

headroom_t **vect_complex_s16_mag**(int16_t a[], const int16_t b_real[], const int16_t b_imag[], const unsigned length, const *right_shift_t* b_shr, const int16_t *rot_table, const unsigned table_rows)

Compute the magnitude of each element of a complex 16-bit vector.

a[] represents the real 16-bit output mantissa vector $\bar{a}$.

b_real[] and b_imag[] together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is b_real[k], and each $Im\{b_k\}$ is b_imag[k].

Each of the input vectors must begin at a word-aligned address. This operation can be performed safely in-place on inputs b_real[] or b_imag[].

length is the number of elements in each of the vectors.

b_shr is the signed arithmetic right-shift applied to elements of $\bar{b}$.

rot_table must point to a pre-computed table of complex vectors used in calculating the magnitudes. table_rows is the number of rows in the table. This library is distributed with a default version of the required rotation table. The following symbols can be used to refer to it in user code:

```
const extern unsigned rot_table16_rows;
const extern complex_s16_t rot_table16[30][4];
```

Faster computation (with reduced precision) can be achieved by generating a smaller version of the table. A python script is provided to generate this table.

Operation Performed

$$v_k \leftarrow b_k \cdot 2^{-b\_shr}$$

$$a_k \leftarrow \sqrt{(Re\{v_k\})^2 + (Im\{v_k\})^2}$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 16-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the real 16-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + b\_shr$.

The function *vect_complex_s16_mag_prepare()* can be used to obtain values for a_exp and b_shr based on the input exponent b_exp and headroom b_hr.

See also:

vect_complex_s16_mag_prepare

Parameters

- **a** − **[out]** Real output vector $\bar{a}$
- **b_real** − **[in]** Real part of complex input vector $\bar{b}$
- **b_imag** − **[in]** Imag part of complex input vector $\bar{b}$
- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- **b_shr** − **[in]** Right-shift appled to $\bar{b}$
- **rot_table** − **[in]** Pre-computed rotation table required for calculating magnitudes
- **table_rows** − **[in]** Number of rows in rot_table

Throws ET_LOAD_STORE
Raised if a, b_real or b_imag is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$.

headroom_t **vect_complex_s16_macc**(int16_t acc_real[], int16_t acc_imag[], const int16_t b_real[], const int16_t b_imag[], const int16_t c_real[], const int16_t c_imag[], const unsigned length, const *right_shift_t* acc_shr, const *right_shift_t* bc_sat)

Multiply one complex 16-bit vector element-wise by another, and add the result to an accumulator.

acc_real[] and acc_imag[] together represent the complex 16-bit accumulator mantissa vector $\bar{a}$. Each $Re\{a_k\}$ is acc_real[k], and each $Im\{a_k\}$ is acc_imag[k].

b_real[] and b_imag[] together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is b_real[k], and each $Im\{b_k\}$ is b_imag[k].

`c_real[]` and `c_imag[]` together represent the complex 16-bit input mantissa vector $\bar{c}$. Each $Re\{c_k\}$ is `c_real[k]`, and each $Im\{c_k\}$ is `c_imag[k]`.

Each of the input vectors must begin at a word-aligned address.

`length` is the number of elements in each of the vectors.

`acc_shr` is the signed arithmetic right-shift applied to the accumulators a_k.

`bc_sat` is the unsigned arithmetic right-shift applied to the product of b_k and c_k before being added to the accumulator.

Operation Performed

$$v_k \leftarrow Re\{b_k\} \cdot Re\{c_k\} - Im\{b_k\} \cdot Im\{c_k\}$$
$$s_k \leftarrow Im\{b_k\} \cdot Re\{c_k\} + Re\{b_k\} \cdot Im\{c_k\}$$
$$\hat{a}_k \leftarrow sat_{16}(a_k \cdot 2^{-acc\_shr})$$
$$Re\{a_k\} \leftarrow sat_{16}(Re\{\hat{a}_k\} + round(sat_{16}(v_k \cdot 2^{-bc\_sat})))$$
$$Im\{a_k\} \leftarrow sat_{16}(Im\{\hat{a}_k\} + round(sat_{16}(s_k \cdot 2^{-bc\_sat})))$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If inputs $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, and input $\bar{a}$ is the accumulator BFP vector $\bar{a} \cdot 2^{a\_exp}$, then the output values of $\bar{a}$ have the exponent $2^{a\_exp+acc\_shr}$.

For accumulation to make sense mathematically, bc_sat must be chosen such that $a\_exp + acc\_shr = b\_exp + c\_exp + bc\_sat$.

The function *vect_complex_s16_macc_prepare()* can be used to obtain values for a_exp, acc_shr and bc_sat based on the input exponents a_exp, b_exp and c_exp and the input headrooms a_hr, b_hr and c_hr.

See also:

vect_complex_s16_macc_prepare

Parameters

- **`acc_real`** – **[inout]** Real part of complex accumulator $\bar{a}$
- **`acc_imag`** – **[inout]** Imaginary aprt of complex accumulator $\bar{a}$
- **`b_real`** – **[in]** Real part of complex input vector $\bar{b}$
- **`b_imag`** – **[in]** Imaginary part of complex input vector $\bar{b}$
- **`c_real`** – **[in]** Real part of complex input vector $\bar{c}$
- **`c_imag`** – **[in]** Imaginary part of complex input vector $\bar{c}$
- **`length`** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **`acc_shr`** – **[in]** Signed arithmetic right-shift applied to accumulator elements.
- **`bc_sat`** – **[in]** Unsigned arithmetic right-shift applied to the products of elements b_k and c_k

Throws ET_LOAD_STORE
Raised if `acc_real`, `acc_imag`, `b_real`, `b_imag`, `c_real` or `c_imag` is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of the output vector $\bar{a}$

headroom_t **vect_complex_s16_nmacc**(int16_t acc_real[], int16_t acc_imag[], const int16_t b_real[], const int16_t b_imag[], const int16_t c_real[], const int16_t c_imag[], const unsigned length, const *right_shift_t* acc_shr, const *right_shift_t* bc_sat)

Multiply one complex 16-bit vector element-wise by another, and subtract the result from an accumulator.

`acc_real[]` and `acc_imag[]` together represent the complex 16-bit accumulator mantissa vector $\bar{a}$. Each $Re\{a_k\}$ is `acc_real[k]`, and each $Im\{a_k\}$ is `acc_imag[k]`.

`b_real[]` and `b_imag[]` together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is `b_real[k]`, and each $Im\{b_k\}$ is `b_imag[k]`.

`c_real[]` and `c_imag[]` together represent the complex 16-bit input mantissa vector $\bar{c}$. Each $Re\{c_k\}$ is `c_real[k]`, and each $Im\{c_k\}$ is `c_imag[k]`.

Each of the input vectors must begin at a word-aligned address.

`length` is the number of elements in each of the vectors.

`acc_shr` is the signed arithmetic right-shift applied to the accumulators a_k.

`bc_sat` is the unsigned arithmetic right-shift applied to the product of b_k and c_k before being subtracted from the accumulator.

Operation Performed

$$v_k \leftarrow Re\{b_k\} \cdot Re\{c_k\} - Im\{b_k\} \cdot Im\{c_k\}$$
$$s_k \leftarrow Im\{b_k\} \cdot Re\{c_k\} + Re\{b_k\} \cdot Im\{c_k\}$$
$$\hat{a}_k \leftarrow sat_{16}(a_k \cdot 2^{-acc\_shr})$$
$$Re\{a_k\} \leftarrow sat_{16}(Re\{\hat{a}_k\} - round(sat_{16}(v_k \cdot 2^{-bc\_sat})))$$
$$Im\{a_k\} \leftarrow sat_{16}(Im\{\hat{a}_k\} - round(sat_{16}(s_k \cdot 2^{-bc\_sat})))$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If inputs $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, and input $\bar{a}$ is the accumulator BFP vector $\bar{a} \cdot 2^{a\_exp}$, then the output values of $\bar{a}$ have the exponent $2^{a\_exp+acc\_shr}$.

For accumulation to make sense mathematically, bc_sat must be chosen such that $a\_exp + acc\_shr = b\_exp + c\_exp + bc\_sat$.

The function *vect_complex_s16_nmacc_prepare()* can be used to obtain values for a_exp, acc_shr and bc_sat based on the input exponents a_exp, b_exp and c_exp and the input headrooms a_hr, b_hr and c_hr.

See also:

vect_complex_s16_nmacc_prepare

Parameters

- **acc_real** – **[inout]** Real part of complex accumulator $\bar{a}$
- **acc_imag** – **[inout]** Imaginary aprt of complex accumulator $\bar{a}$

- **b_real** – **[in]** Real part of complex input vector $\bar{b}$

- **b_imag** – **[in]** Imaginary part of complex input vector $\bar{b}$

- **c_real** – **[in]** Real part of complex input vector $\bar{c}$

- **c_imag** – **[in]** Imaginary part of complex input vector $\bar{c}$

- **length** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$

- **acc_shr** – **[in]** Signed arithmetic right-shift applied to accumulator elements.

- **bc_sat** – **[in]** Unsigned arithmetic right-shift applied to the products of elements b_k and c_k

Throws ET_LOAD_STORE

Raised if acc_real, acc_imag, b_real, b_imag, c_real or c_imag is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of the output vector $\bar{a}$

headroom_t **vect_complex_s16_conj_macc**(int16_t acc_real[], int16_t acc_imag[], const int16_t b_real[], const int16_t b_imag[], const int16_t c_real[], const int16_t c_imag[], const unsigned length, const *right_shift_t* acc_shr, const *right_shift_t* bc_sat)

Multiply one complex 16-bit vector element-wise by the complex conjugate of another, and add the result to an accumulator.

acc_real[] and acc_imag[] together represent the complex 16-bit accumulator mantissa vector $\bar{a}$. Each $Re\{a_k\}$ is acc_real[k], and each $Im\{a_k\}$ is acc_imag[k].

b_real[] and b_imag[] together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is b_real[k], and each $Im\{b_k\}$ is b_imag[k].

c_real[] and c_imag[] together represent the complex 16-bit input mantissa vector $\bar{c}$. Each $Re\{c_k\}$ is c_real[k], and each $Im\{c_k\}$ is c_imag[k].

Each of the input vectors must begin at a word-aligned address.

length is the number of elements in each of the vectors.

acc_shr is the signed arithmetic right-shift applied to the accumulators a_k.

bc_sat is the unsigned arithmetic right-shift applied to the product of b_k and c_k^* before being added to the accumulator.

Operation Performed

$$v_k \leftarrow Re\{b_k\} \cdot Re\{c_k\} + Im\{b_k\} \cdot Im\{c_k\}$$
$$s_k \leftarrow Im\{b_k\} \cdot Re\{c_k\} - Re\{b_k\} \cdot Im\{c_k\}$$
$$\hat{a}_k \leftarrow sat_{16}(a_k \cdot 2^{-acc\_shr})$$
$$Re\{a_k\} \leftarrow sat_{16}(Re\{\hat{a}_k\} + round(sat_{16}(v_k \cdot 2^{-bc\_sat})))$$
$$Im\{a_k\} \leftarrow sat_{16}(Im\{\hat{a}_k\} + round(sat_{16}(s_k \cdot 2^{-bc\_sat})))$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If inputs $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, and input $\bar{a}$ is the accumulator BFP vector $\bar{a} \cdot 2^{a\_exp}$, then the output values of $\bar{a}$ have the exponent $2^{a\_exp+acc\_shr}$.

For accumulation to make sense mathematically, bc_sat must be chosen such that $a\_exp + acc\_shr = b\_exp + c\_exp + bc\_sat$.

The function *vect_complex_s16_macc_prepare()* can be used to obtain values for a_exp, acc_shr and bc_sat based on the input exponents a_exp, b_exp and c_exp and the input headrooms a_hr, b_hr and c_hr.

See also:

vect_complex_s16_conj_macc_prepare

Parameters

- `acc_real` – **[inout]** Real part of complex accumulator $\bar{a}$
- `acc_imag` – **[inout]** Imaginary aprt of complex accumulator $\bar{a}$
- `b_real` – **[in]** Real part of complex input vector $\bar{b}$
- `b_imag` – **[in]** Imaginary part of complex input vector $\bar{b}$
- `c_real` – **[in]** Real part of complex input vector $\bar{c}$
- `c_imag` – **[in]** Imaginary part of complex input vector $\bar{c}$
- `length` – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- `acc_shr` – **[in]** Signed arithmetic right-shift applied to accumulator elements.
- `bc_sat` – **[in]** Unsigned arithmetic right-shift applied to the products of elements b_k and c_k^*

Throws ET_LOAD_STORE

Raised if `acc_real`, `acc_imag`, `b_real`, `b_imag`, `c_real` or `c_imag` is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of the output vector $\bar{a}$

headroom_t **vect_complex_s16_conj_nmacc**(int16_t acc_real[], int16_t acc_imag[], const int16_t b_real[], const int16_t b_imag[], const int16_t c_real[], const int16_t c_imag[], const unsigned length, const *right_shift_t* acc_shr, const *right_shift_t* bc_sat)

Multiply one complex 16-bit vector element-wise by the complex conjugate of another, and subtract the result from an accumulator.

`acc_real[]` and `acc_imag[]` together represent the complex 16-bit accumulator mantissa vector $\bar{a}$. Each $Re\{a_k\}$ is `acc_real[k]`, and each $Im\{a_k\}$ is `acc_imag[k]`.

`b_real[]` and `b_imag[]` together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is `b_real[k]`, and each $Im\{b_k\}$ is `b_imag[k]`.

`c_real[]` and `c_imag[]` together represent the complex 16-bit input mantissa vector $\bar{c}$. Each $Re\{c_k\}$ is `c_real[k]`, and each $Im\{c_k\}$ is `c_imag[k]`.

Each of the input vectors must begin at a word-aligned address.

`length` is the number of elements in each of the vectors.

`acc_shr` is the signed arithmetic right-shift applied to the accumulators a_k.

`bc_sat` is the unsigned arithmetic right-shift applied to the product of b_k and c_k^* before being subtracted from the accumulator.

Operation Performed

$$v_k \leftarrow Re\{b_k\} \cdot Re\{c_k\} + Im\{b_k\} \cdot Im\{c_k\}$$

$$s_k \leftarrow Im\{b_k\} \cdot Re\{c_k\} - Re\{b_k\} \cdot Im\{c_k\}$$

$$\hat{a}_k \leftarrow sat_{16}(a_k \cdot 2^{-acc\_shr})$$

$$Re\{a_k\} \leftarrow sat_{16}(Re\{\hat{a}_k\} - round(sat_{16}(v_k \cdot 2^{-bc\_sat})))$$

$$Im\{a_k\} \leftarrow sat_{16}(Im\{\hat{a}_k\} - round(sat_{16}(s_k \cdot 2^{-bc\_sat})))$$

$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If inputs $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, and input $\bar{a}$ is the accumulator BFP vector $\bar{a} \cdot 2^{a\_exp}$, then the output values of $\bar{a}$ have the exponent $2^{a\_exp + acc\_shr}$.

For accumulation to make sense mathematically, bc_sat must be chosen such that $a\_exp + acc\_shr = b\_exp + c\_exp + bc\_sat$.

The function *vect_complex_s16_macc_prepare()* can be used to obtain values for a_exp, acc_shr and bc_sat based on the input exponents a_exp, b_exp and c_exp and the input headrooms a_hr, b_hr and c_hr.

See also:

vect_complex_s16_conj_nmacc_prepare

Parameters

- **acc_real** – **[inout]** Real part of complex accumulator $\bar{a}$
- **acc_imag** – **[inout]** Imaginary aprt of complex accumulator $\bar{a}$
- **b_real** – **[in]** Real part of complex input vector $\bar{b}$
- **b_imag** – **[in]** Imaginary part of complex input vector $\bar{b}$
- **c_real** – **[in]** Real part of complex input vector $\bar{c}$
- **c_imag** – **[in]** Imaginary part of complex input vector $\bar{c}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **acc_shr** – **[in]** Signed arithmetic right-shift applied to accumulator elements.
- **bc_sat** – **[in]** Unsigned arithmetic right-shift applied to the products of elements b_k and c_k^*

Throws ET_LOAD_STORE
Raised if `acc_real`, `acc_imag`, `b_real`, `b_imag`, `c_real` or `c_imag` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$

headroom_t **vect_complex_s16_mul**(int16_t a_real[], int16_t a_imag[], const int16_t b_real[], const int16_t b_imag[], const int16_t c_real[], const int16_t c_imag[], const unsigned length, const *right_shift_t* a_shr)

Multiply one complex 16-bit vector element-wise by another.

`a_real[]` and `a_imag[]` together represent the complex 16-bit output mantissa vector $\bar{a}$. Each $Re\{a_k\}$ is `a_real[k]`, and each $Im\{a_k\}$ is `a_imag[k]`.

`b_real[]` and `b_imag[]` together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is `b_real[k]`, and each $Im\{b_k\}$ is `b_imag[k]`.

`c_real[]` and `c_imag[]` together represent the complex 16-bit input mantissa vector $\bar{c}$. Each $Re\{c_k\}$ is `c_real[k]`, and each $Im\{c_k\}$ is `c_imag[k]`.

Each of the input vectors must begin at a word-aligned address. This operation can be performed safely in-place on inputs `b_real[]`, `b_imag[]`, `c_real[]` and `c_imag[]`.

`length` is the number of elements in each of the vectors.

`a_shr` is the unsigned arithmetic right-shift applied to the 32-bit accumulators holding intermediate results.

Operation Performed

$$v_k = \leftarrow Re\{b_k\} \cdot Re\{c_k\} - Im\{b_k\} \cdot Im\{c_k\}$$
$$s_k = \leftarrow Im\{b_k\} \cdot Re\{c_k\} + Re\{b_k\} \cdot Im\{c_k\}$$
$$Re\{a_k\} \leftarrow round(sat_{16}(v_k \cdot 2^{-a\_shr}))$$
$$Im\{a_k\} \leftarrow round(sat_{16}(s_k \cdot 2^{-a\_shr}))$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 16-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$ and c is the complex 16-bit mantissa of floating-point value $c \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + c\_exp + a\_shr$.

The function *vect_complex_s16_mul_prepare()* can be used to obtain values for a_exp and a_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

See also:

vect_complex_s16_mul_prepare

Parameters

- `a_real` – **[out]** Real part of complex output vector $\bar{a}$
- `a_imag` – **[out]** Imaginary aprt of complex output vector $\bar{a}$
- `b_real` – **[in]** Real part of complex input vector $\bar{b}$
- `b_imag` – **[in]** Imaginary part of complex input vector $\bar{b}$
- `c_real` – **[in]** Real part of complex input vector $\bar{c}$
- `c_imag` – **[in]** Imaginary part of complex input vector $\bar{c}$
- `length` – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- `a_shr` – **[in]** Right-shift applied to 32-bit intermediate results.

Throws ET_LOAD_STORE
Raised if `a_real`, `a_imag`, `b_real`, `b_imag`, `c_real` or `c_imag` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$

$headroom\_t$ **`vect_complex_s16_real_mul`**(int16_t a_real[], int16_t a_imag[], const int16_t b_real[], const int16_t b_imag[], const int16_t c_real[], const unsigned length, const $right\_shift\_t$ a_shr)

Multiply a complex 16-bit vector element-wise by a real 16-bit vector.

`a_real[]` and `a_imag[]` together represent the complex 16-bit output mantissa vector $\bar{a}$. Each $Re\{a_k\}$ is `a_real[k]`, and each $Im\{a_k\}$ is `a_imag[k]`.

`b_real[]` and `b_imag[]` together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is `b_real[k]`, and each $Im\{b_k\}$ is `b_imag[k]`.

`c_real[]` represents the real 16-bit input mantissa vector $\bar{c}$.

Each of the input vectors must begin at a word-aligned address. This operation can be performed safely in-place on inputs `b_real[]`, `b_imag[]` and `c_real[]`.

`length` is the number of elements in each of the vectors.

`a_shr` is the unsigned arithmetic right-shift applied to the 32-bit accumulators holding the penultimate results.

Operation Performed

$$v_k =\leftarrow Re\{b_k\} \cdot c_k$$
$$s_k =\leftarrow Im\{b_k\} \cdot c_k$$
$$Re\{a_k\} \leftarrow round(sat_{16}(v_k \cdot 2^{-a\_shr}))$$
$$Im\{a_k\} \leftarrow round(sat_{16}(s_k \cdot 2^{-a\_shr}))$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 16-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$ and c is the complex 16-bit mantissa of floating-point value $c \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + c\_exp + a\_shr$.

The function vect_s16_real_mul_prepare() can be used to obtain values for a_exp and a_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

See also:

vect_complex_s16_real_mul_prepare

Parameters

- **`a_real`** – **[out]** Real part of complex output vector $\bar{a}$
- **`a_imag`** – **[out]** Imaginary aprt of complex output vector $\bar{a}$
- **`b_real`** – **[in]** Real part of complex input vector $\bar{b}$
- **`b_imag`** – **[in]** Imaginary part of complex input vector $\bar{b}$
- **`c_real`** – **[in]** Real part of complex input vector $\bar{c}$
- **`length`** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **`a_shr`** – **[in]** Right-shift applied to 32-bit intermediate results.

Throws ET_LOAD_STORE
Raised if `a_real`, `a_imag`, `b_real`, `b_imag` or `c_real` is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of the output vector $\bar{a}$.

$headroom\_t$ **vect_complex_s16_real_scale**(int16_t a_real[], int16_t a_imag[], const int16_t b_real[], const int16_t b_imag[], const int16_t c, const unsigned length, const $right\_shift\_t$ a_shr)

Multiply a complex 16-bit vector by a real scalar.

`a_real[]` and `a_imag[]` together represent the complex 16-bit output mantissa vector $\bar{a}$. Each $Re\{a_k\}$ is `a_real[k]`, and each $Im\{a_k\}$ is `a_imag[k]`.

`b_real[]` and `b_imag[]` together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is `b_real[k]`, and each $Im\{b_k\}$ is `b_imag[k]`.

Each of the input vectors must begin at a word-aligned address. This operation can be performed safely in-place on inputs `b_real[]` and `b_imag[]`.

`c` is the real 16-bit input mantissa c.

`length` is the number of elements in each of the vectors.

`a_shr` is an unsigned arithmetic right-shift applied to the 32-bit accumulators holding the penultimate results.

Operation Performed

$$v_k =\leftarrow Re\{b_k\} \cdot c$$
$$s_k =\leftarrow Im\{b_k\} \cdot c$$
$$Re\{a_k\} \leftarrow round(sat_{16}(v_k \cdot 2^{-a\_shr}))$$
$$Im\{a_k\} \leftarrow round(sat_{16}(s_k \cdot 2^{-a\_shr}))$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 16-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$ and c is the complex 16-bit mantissa of floating-point value $c \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + c\_exp + a\_shr$.

The function $vect\_complex\_s16\_real\_scale\_prepare()$ can be used to obtain values for a_exp and a_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Parameters

- `a_real` – **[out]** Real part of complex output vector $\bar{a}$
- `a_imag` – **[out]** Imaginary aprt of complex output vector $\bar{a}$
- `b_real` – **[in]** Real part of complex input vector $\bar{b}$
- `b_imag` – **[in]** Imaginary part of complex input vector $\bar{b}$
- `c` – **[in]** Real input scalar c
- `length` – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- `a_shr` – **[in]** Right-shift applied to 32-bit intermediate results.

Throws ET_LOAD_STORE

Raised if `a_real`, `a_imag`, `b_real`, `b_imag` or `c` is not word-aligned (See *Note: Vector Alignment*)

Returns
> Headroom of the output vector $\bar{a}$.

headroom_t **vect_complex_s16_scale**(int16_t a_real[], int16_t a_imag[], const int16_t b_real[], const int16_t b_imag[], const int16_t c_real, const int16_t c_imag, const unsigned length, const *right_shift_t* a_shr)

Multiply a complex 16-bit vector by a complex 16-bit scalar.

`a_real[]` and `a_imag[]` together represent the complex 16-bit output mantissa vector $\bar{a}$. Each $Re\{a_k\}$ is `a_real[k]`, and each $Im\{a_k\}$ is `a_imag[k]`.

`b_real[]` and `b_imag[]` together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is `b_real[k]`, and each $Im\{b_k\}$ is `b_imag[k]`.

Each of the input vectors must begin at a word-aligned address. This operation can be performed safely in-place on inputs `b_real[]` and `b_imag[]`.

`c_real` and `c_imag` are the real and imaginary parts of the complex 16-bit input mantissa c.

`length` is the number of elements in each of the vectors.

`a_shr` is the unsigned arithmetic right-shift applied to the 32-bit accumulators holding the penultimate results.

Operation Performed

$$v_k = \leftarrow Re\{b_k\} \cdot Re\{c\} - Im\{b_k\} \cdot Im\{c\}$$
$$s_k = \leftarrow Im\{b_k\} \cdot Re\{c\} + Re\{b_k\} \cdot Im\{c\}$$
$$Re\{a_k\} \leftarrow round(sat_{16}(v_k \cdot 2^{-a\_shr}))$$
$$Im\{a_k\} \leftarrow round(sat_{16}(s_k \cdot 2^{-a\_shr}))$$
$$\text{for } k \in 0 \ ... \ (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 16-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$ and c is the complex 16-bit mantissa of floating-point value $c \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + c\_exp + a\_shr$.

The function *vect_complex_s16_scale_prepare()* can be used to obtain values for a_exp and a_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Parameters
- **a_real** – **[out]** Real part of complex output vector $\bar{a}$
- **a_imag** – **[out]** Imaginary aprt of complex output vector $\bar{a}$
- **b_real** – **[in]** Real part of complex input vector $\bar{b}$
- **b_imag** – **[in]** Imaginary part of complex input vector $\bar{b}$
- **c_real** – **[in]** Real part of complex input scalar c
- **c_imag** – **[in]** Imaginary part of complex input scalar c
- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- **a_shr** – **[in]** Right-shift applied to 32-bit intermediate results

Throws ET_LOAD_STORE
> Raised if `a_real`, `a_imag`, `b_real`, `b_imag`, `c_real` or `c_imag` is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of the output vector $\bar{a}$.

void **vect_complex_s16_set**(int16_t a_real[], int16_t a_imag[], const int16_t b_real, const int16_t b_imag, const unsigned length)

Set each element of a complex 16-bit vector to a specified value.

`a_real[]` and `a_imag[]` together represent the complex 16-bit output mantissa vector $\bar{a}$. Each $Re\{a_k\}$ is `a_real[k]`, and each $Im\{a_k\}$ is `a_imag[k]`. Each must begin at a word-aligned address.

`b_real` and `b_imag` are the real and imaginary parts of the complex 16-bit input mantissa b. Each `a_real[k]` will be set to `b_real`. Each `a_imag[k]` will be set to `b_imag`.

`length` is the number of elements in `a_real[]` and `a_imag[]`.

Operation Performed

$$Re\{a_k\} \leftarrow Re\{b\}$$
$$Im\{a_k\} \leftarrow Im\{b\}$$
$$\text{for } k \in 0 \ ... \ (length - 1)$$

Block Floating-Point

If b is the mantissa of floating-point value $b \cdot 2^{b\_exp}$, then the output vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Parameters

- **a_real** – **[out]** Real part of complex output vector $\bar{a}$
- **a_imag** – **[out]** Imaginary aprt of complex output vector $\bar{a}$
- **b_real** – **[in]** Real part of complex input scalar b
- **b_imag** – **[in]** Imaginary part of complex input scalar b
- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE

Raised if `a_real` or `a_imag` is not word-aligned (See *Note: Vector Alignment*)

headroom_t **vect_complex_s16_shl**(int16_t a_real[], int16_t a_imag[], const int16_t b_real[], const int16_t b_imag[], const unsigned length, const *left_shift_t* b_shl)

Left-shift each element of a complex 16-bit vector by a specified number of bits.

`a_real[]` and `a_imag[]` together represent the complex 16-bit output mantissa vector $\bar{a}$. Each $Re\{a_k\}$ is `a_real[k]`, and each $Im\{a_k\}$ is `a_imag[k]`.

`b_real[]` and `b_imag[]` together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is `b_real[k]`, and each $Im\{b_k\}$ is `b_imag[k]`.

Each of the input vectors must begin at a word-aligned address. This operation can be performed safely in-place on inputs `b_real[]` and `b_imag[]`.

`length` is the number of elements in $\bar{a}$ and $\bar{b}$.

`b_shl` is the signed arithmetic left-shift applied to each element of $\bar{b}$.

Operation Performed

$$Re\{a_k\} \leftarrow sat_{16}(\lfloor Re\{b_k\} \cdot 2^{b\_shl} \rfloor)$$
$$Im\{a_k\} \leftarrow sat_{16}(\lfloor Im\{b_k\} \cdot 2^{b\_shl} \rfloor)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 16-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the complex 16-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $\bar{a} = \bar{b} \cdot 2^{b\_shl}$ and $a\_exp = b\_exp$.

Parameters

- **`a_real`** – **[out]** Real part of complex output vector $\bar{a}$

- **`a_imag`** – **[out]** Imaginary aprt of complex output vector $\bar{a}$

- **`b_real`** – **[in]** Real part of complex input vector $\bar{b}$

- **`b_imag`** – **[in]** Imaginary part of complex input vector $\bar{b}$

- **`length`** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

- **`b_shl`** – **[in]** Left-shift applied to $\bar{b}$

Throws ET_LOAD_STORE
Raised if `a_real`, `a_imag`, `b_real` or `b_imag` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$

headroom_t **vect_complex_s16_shr**(int16_t a_real[], int16_t a_imag[], const int16_t b_real[], const int16_t b_imag[], const unsigned length, const *right_shift_t* b_shr)

Right-shift each element of a complex 16-bit vector by a specified number of bits.

`a_real[]` and `a_imag[]` together represent the complex 16-bit output mantissa vector $\bar{a}$. Each $Re\{a_k\}$ is `a_real[k]`, and each $Im\{a_k\}$ is `a_imag[k]`.

`b_real[]` and `b_imag[]` together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is `b_real[k]`, and each $Im\{b_k\}$ is `b_imag[k]`.

Each of the input vectors must begin at a word-aligned address. This operation can be performed safely in-place on inputs `b_real[]` and `b_imag[]`.

`length` is the number of elements in $\bar{a}$ and $\bar{b}$.

`b_shr` is the signed arithmetic right-shift applied to each element of $\bar{b}$.

Operation Performed

$$Re\{a_k\} \leftarrow sat_{16}(\lfloor Re\{b_k\} \cdot 2^{-b\_shr} \rfloor)$$
$$Im\{a_k\} \leftarrow sat_{16}(\lfloor Im\{b_k\} \cdot 2^{-b\_shr} \rfloor)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 16-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the complex 16-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $\bar{a} = \bar{b} \cdot 2^{-b\_shr}$ and $a\_exp = b\_exp$.

Parameters

- **`a_real`** – **[out]** Real part of complex output vector $\bar{a}$

- **`a_imag`** – **[out]** Imaginary aprt of complex output vector $\bar{a}$

- **`b_real`** – **[in]** Real part of complex input vector $\bar{b}$

- **`b_imag`** – **[in]** Imaginary part of complex input vector $\bar{b}$

- **`length`** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

- **`b_shr`** – **[in]** Right-shift applied to $\bar{b}$

Throws ET_LOAD_STORE

Raised if `a_real`, `a_imag`, `b_real` or `b_imag` is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of the output vector $\bar{a}$

headroom_t **vect_complex_s16_squared_mag**(int16_t a[], const int16_t b_real[], const int16_t b_imag[], const unsigned length, const *right_shift_t* a_shr)

Get the squared magnitudes of elements of a complex 16-bit vector.

`a[]` represents the real 16-bit output mantissa vector $\bar{a}$.

`b_real[]` and `b_imag[]` together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is `b_real[k]`, and each $Im\{b_k\}$ is `b_imag[k]`.

Each of the input vectors must begin at a word-aligned address.

`length` is the number of elements in each of the vectors.

`a_shr` is the unsigned arithmetic right-shift applied to the 32-bit accumulators holding the penultimate results.

Operation Performed

$$a_k \leftarrow ((Re\{b_k'\})^2 + (Im\{b_k'\})^2) \cdot 2^{-a\_shr}$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 16-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the real 16-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = 2 \cdot b\_exp + a\_shr$.

The function *vect_complex_s16_squared_mag_prepare()* can be used to obtain values for a_exp and a_shr based on the input exponent b_exp and headroom b_hr.

See also:

vect_complex_s16_squared_mag_prepare

Parameters

- **`a`** – **[out]** Real output vector $\bar{a}$

- **`b_real`** – **[in]** Real part of complex input vector $\bar{b}$

- **`b_imag`** – **[in]** Imaginary part of complex input vector $\bar{b}$

- **`length`** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

- **a_shr** – **[in]** Right-shift appled to 32-bit intermediate results

Throws ET_LOAD_STORE

Raised if a, b_real or b_imag is not word-aligned (See *Note: Vector Alignment*)

headroom_t **vect_complex_s16_sub**(int16_t a_real[], int16_t a_imag[], const int16_t b_real[], const int16_t b_imag[], const int16_t c_real[], const int16_t c_imag[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Subtract one complex 16-bit vector from another.

a_real[] and a_imag[] together represent the complex 16-bit output mantissa vector $\bar{a}$. Each $Re\{a_k\}$ is a_real[k], and each $Im\{a_k\}$ is a_imag[k].

b_real[] and b_imag[] together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is b_real[k], and each $Im\{b_k\}$ is b_imag[k].

c_real[] and c_imag[] together represent the complex 16-bit input mantissa vector $\bar{c}$. Each $Re\{c_k\}$ is c_real[k], and each $Im\{c_k\}$ is c_imag[k].

Each of the input vectors must begin at a word-aligned address. This operation can be performed safely in-place on inputs b_real[], b_imag[], c_real[] and c_imag[].

length is the number of elements in each of the vectors.

b_shr and c_shr are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k \leftarrow sat_{16}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k \leftarrow sat_{16}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$Re\{a_k\} \leftarrow Re\{b'_k\} - Re\{c'_k\}$$
$$Im\{a_k\} \leftarrow Im\{b'_k\} - Im\{c'_k\}$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the complex 16-bit mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the complex 16-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$.

In this case, b_shr and c_shr **must** be chosen so that $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$. Adding or subtracting mantissas only makes sense if they are associated with the same exponent.

The function *vect_complex_s16_sub_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

See also:

vect_complex_s16_sub_prepare

Parameters

- **a_real** – **[out]** Real part of complex output vector $\bar{a}$
- **a_imag** – **[out]** Imaginary aprt of complex output vector $\bar{a}$

- **b_real** – **[in]** Real part of complex input vector $\bar{b}$

- **b_imag** – **[in]** Imaginary part of complex input vector $\bar{b}$

- **c_real** – **[in]** Real part of complex input vector $\bar{c}$

- **c_imag** – **[in]** Imaginary part of complex input vector $\bar{c}$

- **length** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$

- **b_shr** – **[in]** Right-shift applied to $\bar{b}$

- **c_shr** – **[in]** Right-shift applied to $\bar{c}$

Throws ET_LOAD_STORE
Raised if `a_real`, `a_imag`, `b_real`, `b_imag`, `c_real` or `c_imag` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of output vector $\bar{a}$.

complex_s32_t **vect_complex_s16_sum**(const int16_t b_real[], const int16_t b_imag[], const unsigned length)

Get the sum of elements of a complex 16-bit vector.

`b_real[]` and `b_imag[]` together represent the complex 16-bit input mantissa vector $\bar{b}$, and must both begin at a word-aligned address. Each $Re\{b_k\}$ is `b_real[k]`, and each $Im\{b_k\}$ is `b_imag[k]`.

`length` is the number of elements in $\bar{b}$.

Operation Performed

$$Re\{a\} \leftarrow \sum_{k=0}^{length-1} (Re\{b_k\})$$

$$Im\{a\} \leftarrow \sum_{k=0}^{length-1} (Im\{b_k\})$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the returned value a is the complex 32-bit mantissa of floating-point value $a \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Parameters

- **b_real** – **[in]** Real part of complex input vector $\bar{b}$

- **b_imag** – **[in]** Imaginary part of complex input vector $\bar{b}$

- **length** – **[in]** Number of elements in vector $\bar{b}$.

Throws ET_LOAD_STORE
Raised if `b_real` or `b_imag` is not word-aligned (See *Note: Vector Alignment*)

Returns
a, the 32-bit complex sum of elements in $\bar{b}$.

void **vect_complex_s16_to_vect_complex_s32**(*complex_s32_t* a[], const int16_t b_real[], const int16_t b_imag[], const unsigned length)

Convert a complex 16-bit vector into a complex 32-bit vector.

`a[]` represents the complex 32-bit output vector $\bar{a}$. It must begin at a double word (8-byte) aligned address.

`b_real[]` and `b_imag[]` together represent the complex 16-bit input mantissa vector $\bar{b}$. Each $Re\{b_k\}$ is `b_real[k]`, and each $Im\{b_k\}$ is `b_imag[k]`.

The parameter `length` is the number of elements in each of the vectors.

`length` is the number of elements in each of the vectors.

Operation Performed

$$Re\{a_k\} \leftarrow Re\{b_k\}$$
$$Im\{a_k\} \leftarrow Im\{b_k\}$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 16-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the complex 32-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Notes

- The headroom of output vector $\bar{a}$ is not returned by this function. The headroom of the output is always 16 bits greater than the headroom of the input.

Parameters

- **a** – **[out]** Complex output vector $\bar{a}$.
- **b_real** – **[in]** Real part of complex input vector $\bar{b}$.
- **b_imag** – **[in]** Imaginary part of complex input vector $\bar{b}$.
- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE
Raised if `a` is not double-word-aligned (See *Note: Vector Alignment*)

4.7.7 Complex 32-Bit Vector API

group **vect_complex_s32_api**

Functions

$headroom\_t$ **`vect_complex_s32_add`**($complex\_s32\_t$ a[], const $complex\_s32\_t$ b[], const $complex\_s32\_t$ c[], const unsigned length, const $right\_shift\_t$ b_shr, const $right\_shift\_t$ c_shr)

Add one complex 32-bit vector to another.

`a[]`, `b[]` and `c[]` represent the complex 32-bit mantissa vectors $\bar{a}, \bar{b}$ and $\bar{c}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on `b[]` or `c[]`.

`length` is the number of elements in each of the vectors.

`b_shr` and `c_shr` are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k \leftarrow sat_{32}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$Re\{a_k\} \leftarrow Re\{b'_k\} + Re\{c'_k\}$$
$$Im\{a_k\} \leftarrow Im\{b'_k\} + Im\{c'_k\}$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the complex 32-bit mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the complex 32-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$.

In this case, b_shr and c_shr **must** be chosen so that $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$. Adding or subtracting mantissas only makes sense if they are associated with the same exponent.

The function $vect\_complex\_s32\_add\_prepare()$ can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

See also:

$vect\_complex\_s32\_add\_prepare$

Parameters

- **a** − **[out]** Complex output vector $\bar{a}$
- **b** − **[in]** Complex input vector $\bar{b}$
- **c** − **[in]** Complex input vector $\bar{c}$
- **length** − **[in]** Number of elements in vectors $\bar{a}, \bar{b}$ and $\bar{c}$
- **b_shr** − **[in]** Right-shift applied to $\bar{b}$
- **c_shr** − **[in]** Right-shift applied to $\bar{c}$

Throws ET_LOAD_STORE

Raised if `a`, `b` or `c` is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of output vector $\bar{a}$.

$headroom\_t$ **vect_complex_s32_add_scalar**(*complex_s32_t* a[], const *complex_s32_t* b[], const *complex_s32_t* c, const unsigned length, const *right_shift_t* b_shr)

Add a scalar to a complex 32-bit vector.

a[] and b[] represent the complex 32-bit mantissa vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[].

c is the complex scalar c to be added to each element of $\bar{b}$.

length is the number of elements in each of the vectors.

b_shr is the signed arithmetic right-shift applied to each element of $\bar{b}$.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$Re\{a_k\} \leftarrow Re\{b'_k\} + Re\{c\}$$
$$Im\{a_k\} \leftarrow Im\{b'_k\} + Im\{c\}$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If elements of $\bar{b}$ are the complex mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, and c is the mantissa of floating-point value $c \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$.

In this case, b_shr and c_shr **must** be chosen so that $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$. Adding or subtracting mantissas only makes sense if they are associated with the same exponent.

The function *vect_complex_s32_add_scalar_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Note that c_shr is an output of *vect_complex_s32_add_scalar_prepare()*, but is not a parameter to this function. The c_shr produced by *vect_complex_s32_add_scalar_prepare()* is to be applied by the user, and the result passed as input c.

See also:

vect_complex_s32_add_scalar_prepare

Parameters

- **a** − **[out]** Complex output vector $\bar{a}$
- **b** − **[in]** Complex input vector $\bar{b}$
- **c** − **[in]** Complex input scalar c
- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- **b_shr** − **[in]** Right-shift applied to $\bar{b}$

Throws ET_LOAD_STORE
Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of output vector $\bar{a}$.

headroom_t **vect_complex_s32_conj_mul**(*complex_s32_t* a[], const *complex_s32_t* b[], const *complex_s32_t* c[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Multiply one complex 32-bit vector element-wise by the complex conjugate of another.

a[], b[] and c[] represent the 32-bit mantissa vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[] or c[].

length is the number of elements in each of the vectors.

b_shr and c_shr are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k \leftarrow sat_{32}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$Re\{a_k\} \leftarrow (Re\{b'_k\} \cdot Re\{c'_k\} + Im\{b'_k\} \cdot Im\{c'_k\}) \cdot 2^{-30}$$
$$Im\{a_k\} \leftarrow (Im\{b'_k\} \cdot Re\{c'_k\} - Re\{b'_k\} \cdot Im\{c'_k\}) \cdot 2^{-30}$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 32-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$ and c is the complex 32-bit mantissa of floating-point value $c \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + c\_exp + a\_shr$.

The function *vect_complex_s32_conj_mul_prepare()* can be used to obtain values for a_exp and a_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

See also:

vect_complex_s32_conj_mul_prepare

Parameters

- **a** − **[out]** Complex output vector $\bar{a}$
- **b** − **[in]** Complex input vector $\bar{b}$
- **c** − **[in]** Complex input vector $\bar{c}$
- **length** − **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **b_shr** − **[in]** Right-shift applied to elements of $\bar{b}$.
- **c_shr** − **[in]** Right-shift applied to elements of $\bar{c}$.

Throws ET_LOAD_STORE
Raised if a, b or c is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$

headroom_t **vect_complex_s32_headroom**(const *complex_s32_t* x[], const unsigned length)

Calculate the headroom of a complex 32-bit array.

The headroom of an N-bit integer is the number of bits that the integer's value may be left-shifted without any information being lost. Equivalently, it is one less than the number of leading sign bits.

The headroom of a `complex_s32_t` struct is the minimum of the headroom of each of its 32-bit fields, `re` and `im`.

The headroom of a `complex_s32_t` array is the minimum of the headroom of each of its `complex_s32_t` elements.

This function efficiently traverses the elements of $\bar{x}$ to determine its headroom.

`x[]` represents the complex 32-bit vector $\bar{x}$. `x[]` must begin at a word-aligned address.

`length` is the number of elements in `x[]`.

Operation Performed

$$min\{HR_{32}\left(x_0\right), HR_{32}\left(x_1\right), ..., HR_{32}\left(x_{length-1}\right)\}$$

See also:

vect_s16_headroom, *vect_s32_headroom*, *vect_complex_s16_headroom*

> **Parameters**
>
> - **x** – **[in]** Complex input vector $\bar{x}$
>
> - **length** – **[in]** Number of elements in $\bar{x}$
>
> **Throws ET_LOAD_STORE**
> Raised if `x` is not word-aligned (See *Note: Vector Alignment*)
>
> **Returns**
> Headroom of vector $\bar{x}$

headroom_t **vect_complex_s32_macc**(*complex_s32_t* acc[], const *complex_s32_t* b[], const *complex_s32_t* c[], const unsigned length, const *right_shift_t* acc_shr, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Multiply one complex 32-bit vector element-wise by another, and add the result to an accumulator.

`acc[]` represents the complex 32-bit accumulator mantissa vector $\bar{a}$. Each a_k is `acc[k]`.

`b[]` and `c[]` represent the complex 32-bit input mantissa vectors $\bar{b}$ and $\bar{c}$, where each b_k is `b[k]` and each c_k is `c[k]`.

Each of the input vectors must begin at a word-aligned address.

`length` is the number of elements in each of the vectors.

`acc_shr`, `b_shr` and `c_shr` are the signed arithmetic right-shifts applied to input elements a_k, b_k and c_k.

Operation Performed

$$\tilde{b}_k \leftarrow sat_{32}(b_k \cdot 2^{-b\_shr})$$

$$\tilde{c}_k \leftarrow sat_{32}(c_k \cdot 2^{-c\_shr})$$

$$\tilde{a}_k \leftarrow sat_{32}(a_k \cdot 2^{-acc\_shr})$$

$$v_k \leftarrow round(sat_{32}((Re\{\tilde{b}_k\} \cdot Re\{\tilde{c}_k\} - Im\{\tilde{b}_k\} \cdot Im\{\tilde{c}_k\}) \cdot 2^{-30}))$$

$$s_k \leftarrow round(sat_{32}((Im\{\tilde{b}_k\} \cdot Re\{\tilde{c}_k\} + Re\{\tilde{b}_k\} \cdot Im\{\tilde{c}_k\}) \cdot 2^{-30}))$$

$$Re\{a_k\} \leftarrow sat_{32}(Re\{\tilde{a}_k\} + v_k)$$

$$Im\{a_k\} \leftarrow sat_{32}(Im\{\tilde{a}_k\} + s_k)$$

$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If inputs $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, and input $\bar{a}$ is the accumulator BFP vector $\bar{a} \cdot 2^{a\_exp}$, then the output values of $\bar{a}$ have the exponent $2^{a\_exp + acc\_shr}$.

For accumulation to make sense mathematically, bc_sat must be chosen such that $a\_exp + acc\_shr = b\_exp + c\_exp + b\_shr + c\_shr$.

The function *vect_complex_s32_macc_prepare()* can be used to obtain values for a_exp, acc_shr, b_shr and c_shr based on the input exponents a_exp, b_exp and c_exp and the input headrooms a_hr, b_hr and c_hr.

See also:

vect_complex_s32_macc_prepare

Parameters

- `acc` − **[inout]** Complex accumulator $\bar{a}$
- `b` − **[in]** Complex input vector $\bar{b}$
- `c` − **[in]** Complex input vector $\bar{c}$
- `length` − **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- `acc_shr` − **[in]** Signed arithmetic right-shift applied to accumulator elements.
- `b_shr` − **[in]** Signed arithmetic right-shift applied to elements of $\bar{b}$
- `c_shr` − **[in]** Signed arithmetic right-shift applied to elements of $\bar{c}$

Throws ET_LOAD_STORE
Raised if `acc`, `b` or `c` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$

headroom_t **vect_complex_s32_nmacc**(*complex_s32_t* acc[], const *complex_s32_t* b[], const *complex_s32_t* c[], const unsigned length, const *right_shift_t* acc_shr, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Multiply one complex 32-bit vector element-wise by another, and subtract the result from an accumulator.

`acc[]` represents the complex 32-bit accumulator mantissa vector $\bar{a}$. Each a_k is `acc[k]`.

`b[]` and `c[]` represent the complex 32-bit input mantissa vectors $\bar{b}$ and $\bar{c}$, where each b_k is `b[k]` and each c_k is `c[k]`.

Each of the input vectors must begin at a word-aligned address.

`length` is the number of elements in each of the vectors.

`acc_shr`, `b_shr` and `c_shr` are the signed arithmetic right-shifts applied to input elements a_k, b_k and c_k.

Operation Performed

$$\tilde{b}_k \leftarrow sat_{32}(b_k \cdot 2^{-b\_shr})$$
$$\tilde{c}_k \leftarrow sat_{32}(c_k \cdot 2^{-c\_shr})$$
$$\tilde{a}_k \leftarrow sat_{32}(a_k \cdot 2^{-acc\_shr})$$
$$v_k \leftarrow round(sat_{32}((Re\{\tilde{b}_k\} \cdot Re\{\tilde{c}_k\} - Im\{\tilde{b}_k\} \cdot Im\{\tilde{c}_k\}) \cdot 2^{-30}))$$
$$s_k \leftarrow round(sat_{32}((Im\{\tilde{b}_k\} \cdot Re\{\tilde{c}_k\} + Re\{\tilde{b}_k\} \cdot Im\{\tilde{c}_k\}) \cdot 2^{-30}))$$
$$Re\{a_k\} \leftarrow sat_{32}(Re\{\tilde{a}_k\} - v_k)$$
$$Im\{a_k\} \leftarrow sat_{32}(Im\{\tilde{a}_k\} - s_k)$$
$$\text{for } k \in 0 \ ... \ (length - 1)$$

Block Floating-Point

If inputs $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, and input $\bar{a}$ is the accumulator BFP vector $\bar{a} \cdot 2^{a\_exp}$, then the output values of $\bar{a}$ have the exponent $2^{a\_exp+acc\_shr}$.

For accumulation to make sense mathematically, bc_sat must be chosen such that $a\_exp + acc\_shr = b\_exp + c\_exp + b\_shr + c\_shr$.

The function *vect_complex_s32_macc_prepare()* can be used to obtain values for a_exp, acc_shr, b_shr and c_shr based on the input exponents a_exp, b_exp and c_exp and the input headrooms a_hr, b_hr and c_hr.

See also:

vect_complex_s32_nmacc_prepare

Parameters

- **acc** − **[inout]** Complex accumulator $\bar{a}$
- **b** − **[in]** Complex input vector $\bar{b}$
- **c** − **[in]** Complex input vector $\bar{c}$
- **length** − **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **acc_shr** − **[in]** Signed arithmetic right-shift applied to accumulator elements.
- **b_shr** − **[in]** Signed arithmetic right-shift applied to elements of $\bar{b}$
- **c_shr** − **[in]** Signed arithmetic right-shift applied to elements of $\bar{c}$

Throws ET_LOAD_STORE
Raised if `acc`, `b` or `c` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$

$headroom\_t$ **vect_complex_s32_conj_macc**($complex\_s32\_t$ acc[], const $complex\_s32\_t$ b[], const $complex\_s32\_t$ c[], const unsigned length, const $right\_shift\_t$ acc_shr, const $right\_shift\_t$ b_shr, const $right\_shift\_t$ c_shr)

Multiply one complex 32-bit vector element-wise by the complex conjugate of another, and add the result to an accumulator.

`acc[]` represents the complex 32-bit accumulator mantissa vector $\bar{a}$. Each a_k is `acc[k]`.

`b[]` and `c[]` represent the complex 32-bit input mantissa vectors $\bar{b}$ and $\bar{c}$, where each b_k is `b[k]` and each c_k is `c[k]`.

Each of the input vectors must begin at a word-aligned address.

`length` is the number of elements in each of the vectors.

`acc_shr`, `b_shr` and `c_shr` are the signed arithmetic right-shifts applied to input elements a_k, b_k and c_k.

Operation Performed

$$\tilde{b}_k \leftarrow sat_{32}(b_k \cdot 2^{-b\_shr})$$
$$\tilde{c}_k \leftarrow sat_{32}(c_k \cdot 2^{-c\_shr})$$
$$\tilde{a}_k \leftarrow sat_{32}(a_k \cdot 2^{-acc\_shr})$$
$$v_k \leftarrow round(sat_{32}((Re\{\tilde{b}_k\} \cdot Re\{\tilde{c}_k\} + Im\{\tilde{b}_k\} \cdot Im\{\tilde{c}_k\}) \cdot 2^{-30}))$$
$$s_k \leftarrow round(sat_{32}((Im\{\tilde{b}_k\} \cdot Re\{\tilde{c}_k\} - Re\{\tilde{b}_k\} \cdot Im\{\tilde{c}_k\}) \cdot 2^{-30}))$$
$$Re\{a_k\} \leftarrow sat_{32}(Re\{\tilde{a}_k\} + v_k)$$
$$Im\{a_k\} \leftarrow sat_{32}(Im\{\tilde{a}_k\} + s_k)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If inputs $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, and input $\bar{a}$ is the accumulator BFP vector $\bar{a} \cdot 2^{a\_exp}$, then the output values of $\bar{a}$ have the exponent $2^{a\_exp+acc\_shr}$.

For accumulation to make sense mathematically, bc_sat must be chosen such that $a\_exp + acc\_shr = b\_exp + c\_exp + b\_shr + c\_shr$.

The function *vect_complex_s32_conj_macc_prepare()* can be used to obtain values for a_exp, acc_shr, b_shr and c_shr based on the input exponents a_exp, b_exp and c_exp and the input headrooms a_hr, b_hr and c_hr.

See also:

vect_complex_s32_conj_macc_prepare

Parameters

- **acc** − **[inout]** Complex accumulator $\bar{a}$

- **b** − **[in]** Complex input vector $\bar{b}$

- **c** − **[in]** Complex input vector $\bar{c}$

- **length** − **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$

- **acc_shr** − **[in]** Signed arithmetic right-shift applied to accumulator elements.

- **b_shr** – **[in]** Signed arithmetic right-shift applied to elements of $\bar{b}$

- **c_shr** – **[in]** Signed arithmetic right-shift applied to elements of $\bar{c}$

Throws ET_LOAD_STORE

Raised if acc, b or c is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of the output vector $\bar{a}$

headroom_t **vect_complex_s32_conj_nmacc**(*complex_s32_t* acc[], const *complex_s32_t* b[], const *complex_s32_t* c[], const unsigned length, const *right_shift_t* acc_shr, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Multiply one complex 32-bit vector element-wise by the complex conjugate of another, and subtract the result from an accumulator.

acc[] represents the complex 32-bit accumulator mantissa vector $\bar{a}$. Each a_k is acc[k].

b[] and c[] represent the complex 32-bit input mantissa vectors $\bar{b}$ and $\bar{c}$, where each b_k is b[k] and each c_k is c[k].

Each of the input vectors must begin at a word-aligned address.

length is the number of elements in each of the vectors.

acc_shr, b_shr and c_shr are the signed arithmetic right-shifts applied to input elements a_k, b_k and c_k.

Operation Performed

$$\tilde{b}_k \leftarrow sat_{32}(b_k \cdot 2^{-b\_shr})$$
$$\tilde{c}_k \leftarrow sat_{32}(c_k \cdot 2^{-c\_shr})$$
$$\tilde{a}_k \leftarrow sat_{32}(a_k \cdot 2^{-acc\_shr})$$
$$v_k \leftarrow round(sat_{32}((Re\{\tilde{b}_k\} \cdot Re\{\tilde{c}_k\} + Im\{\tilde{b}_k\} \cdot Im\{\tilde{c}_k\}) \cdot 2^{-30}))$$
$$s_k \leftarrow round(sat_{32}((Im\{\tilde{b}_k\} \cdot Re\{\tilde{c}_k\} - Re\{\tilde{b}_k\} \cdot Im\{\tilde{c}_k\}) \cdot 2^{-30}))$$
$$Re\{a_k\} \leftarrow sat_{32}(Re\{\tilde{a}_k\} - v_k)$$
$$Im\{a_k\} \leftarrow sat_{32}(Im\{\tilde{a}_k\} - s_k)$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If inputs $\bar{b}$ and $\bar{c}$ are the mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, and input $\bar{a}$ is the accumulator BFP vector $\bar{a} \cdot 2^{a\_exp}$, then the output values of $\bar{a}$ have the exponent $2^{a\_exp+acc\_shr}$.

For accumulation to make sense mathematically, bc_sat must be chosen such that $a\_exp + acc\_shr = b\_exp + c\_exp + b\_shr + c\_shr$.

The function *vect_complex_s32_conj_nmacc_prepare()* can be used to obtain values for a_exp, acc_shr, b_shr and c_shr based on the input exponents a_exp, b_exp and c_exp and the input headrooms a_hr, b_hr and c_hr.

See also:

vect_complex_s32_conj_nmacc_prepare

Parameters

- **acc** – **[inout]** Complex accumulator $\bar{a}$

- **b** – **[in]** Complex input vector $\bar{b}$

- **c** – **[in]** Complex input vector $\bar{c}$

- **length** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$

- **acc_shr** – **[in]** Signed arithmetic right-shift applied to accumulator elements.

- **b_shr** – **[in]** Signed arithmetic right-shift applied to elements of $\bar{b}$

- **c_shr** – **[in]** Signed arithmetic right-shift applied to elements of $\bar{c}$

Throws ET_LOAD_STORE
Raised if `acc`, `b` or `c` is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$

headroom_t **vect_complex_s32_mag**(int32_t a[], const *complex_s32_t* b[], const unsigned length, const *right_shift_t* b_shr, const *complex_s32_t* *rot_table, const unsigned table_rows)

Compute the magnitude of each element of a complex 32-bit vector.

`a[]` represents the real 32-bit output mantissa vector $\bar{a}$.

`b[]` represents the complex 32-bit input mantissa vector $\bar{b}$.

`a[]` and `b[]` must each begin at a word-aligned address.

`length` is the number of elements in each of the vectors.

`b_shr` is the signed arithmetic right-shift applied to elements of $\bar{b}$.

`rot_table` must point to a pre-computed table of complex vectors used in calculating the magnitudes. `table_rows` is the number of rows in the table. This library is distributed with a default version of the required rotation table. The following symbols can be used to refer to it in user code:

```
const extern unsigned rot_table32_rows;
const extern complex_s32_t rot_table32[30][4];
```

Faster computation (with reduced precision) can be achieved by generating a smaller version of the table. A python script is provided to generate this table.

Todo:
Point to documentation page on generating this table.

Operation Performed

$$v_k \leftarrow b_k \cdot 2^{-b\_shr}$$
$$a_k \leftarrow \sqrt{(Re\{v_k\})^2 + (Im\{v_k\})^2}$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 32-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the real 32-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + b\_shr$.

The function *vect_complex_s32_mag_prepare()* can be used to obtain values for a_exp and b_shr based on the input exponent b_exp and headroom b_hr.

Parameters

- **a** – **[out]** Real output vector $\bar{a}$
- **b** – **[in]** Complex input vector $\bar{b}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- **b_shr** – **[in]** Right-shift appled to $\bar{b}$
- **rot_table** – **[in]** Pre-computed rotation table required for calculating magnitudes
- **table_rows** – **[in]** Number of rows in `rot_table`

Throws ET_LOAD_STORE
Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$.

headroom_t **vect_complex_s32_mul**(*complex_s32_t* a[], const *complex_s32_t* b[], const *complex_s32_t* c[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Multiply one complex 32-bit vector element-wise by another.

a[], b[] and c[] represent the 32-bit mantissa vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[] or c[].

length is the number of elements in each of the vectors.

b_shr and c_shr are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k \leftarrow sat_{32}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$Re\{a_k\} \leftarrow (Re\{b'_k\} \cdot Re\{c'_k\} - Im\{b'_k\} \cdot Im\{c'_k\}) \cdot 2^{-30}$$
$$Im\{a_k\} \leftarrow (Im\{b'_k\} \cdot Re\{c'_k\} + Re\{b'_k\} \cdot Im\{c'_k\}) \cdot 2^{-30}$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 32-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\text{-}exp}$ and c is the complex 32-bit mantissa of floating-point value $c \cdot 2^{c\text{-}exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\text{-}exp}$, where $a\_exp = b\_exp + c\_exp + b\_shr + c\_shr$.

The function *vect_complex_s32_mul_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Parameters

- **a** – **[out]** Complex output vector $\bar{a}$

- **b** – **[in]** Complex input vector $\bar{b}$

- **c** – **[in]** Complex input vector $\bar{c}$

- `length` – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$, and $\bar{c}$

- `b_shr` – **[in]** Right-shift appled to $\bar{b}$

- `c_shr` – **[in]** Right-shift appled to $\bar{c}$

Throws ET_LOAD_STORE
Raised if a, b or c is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$

headroom_t **vect_complex_s32_real_mul**(*complex_s32_t* a[], const *complex_s32_t* b[], const int32_t c[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Multiply a complex 32-bit vector element-wise by a real 32-bit vector.

a[] and b[] represent the complex 32-bit mantissa vectors $\bar{a}$ and $\bar{b}$ respectively.

c[] represents the real 32-bit mantissa vector $\bar{c}$.

a[], b[], and c[] each must begin at a word-aligned address. This operation can be performed safely in-place on b[].

length is the number of elements in each of the vectors.

b_shr and c_shr are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k \leftarrow sat_{32}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$Re\{a_k\} \leftarrow (Re\{b'_k\} \cdot c'_k) \cdot 2^{-30}$$
$$Im\{a_k\} \leftarrow (Im\{b'_k\} \cdot c'_k) \cdot 2^{-30}$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 32-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$ and c is the complex 32-bit mantissa of floating-point value $c \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + c\_exp + b\_shr + c\_shr$.

The function *vect_complex_s32_real_mul_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

See also:

vect_complex_s32_real_mul_prepare

Parameters

- **a** − **[out]** Complex output vector $\bar{a}$.

- **b** − **[in]** Complex input vector $\bar{b}$.

- **c** − **[in]** Real input vector $\bar{c}$.

- **length** − **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$, and $\bar{c}$.

- **b_shr** − **[in]** Right-shift appled to $\bar{b}$.

- **c_shr** − **[in]** Right-shift appled to $\bar{c}$.

Throws ET_LOAD_STORE
Raised if a, b or c is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$.

headroom_t **vect_complex_s32_real_scale**(*complex_s32_t* a[], const *complex_s32_t* b[], const int32_t c, const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Multiply a complex 32-bit vector by a real scalar.

a[] and b[] represent the complex 32-bit mantissa vectors $\bar{a}$ and $\bar{b}$ respectively.

c represents the real 32-bit scale factor c.

a[] and b[] each must begin at a word-aligned address. This operation can be performed safely in-place on b[].

length is the number of elements in each of the vectors.

b_shr and c_shr are the signed arithmetic right-shift applied to each element of $\bar{b}$ and to c.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$Re\{a_k\} \leftarrow Re\{b'_k\} \cdot c$$
$$Im\{a_k\} \leftarrow Im\{b'_k\} \cdot c$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 16-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$ and c is the complex 16-bit mantissa of floating-point value $c \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + c\_exp + b\_shr + c\_shr$.

The function *vect_complex_s32_real_scale_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Parameters

- **a** − **[out]** Complex output vector $\bar{a}$

- **b** − **[in]** Complex input vector $\bar{b}$

- **c** − **[in]** Complex input vector $\bar{c}$

- **length** − **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$, and $\bar{c}$

- **b_shr** − **[in]** Right-shift applied to $\bar{b}$

- **c_shr** − **[in]** Right-shift applied to c

headroom_t **vect_complex_s32_scale**(*complex_s32_t* a[], const *complex_s32_t* b[], const int32_t c_real, const int32_t c_imag, const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Multiply a complex 32-bit vector by a complex 32-bit scalar.

a[] and b[] represent the complex 32-bit mantissa vectors $\bar{a}$ and $\bar{b}$ respectively.

c represents the complex 32-bit scale factor c.

a[] and b[] each must begin at a word-aligned address. This operation can be performed safely in-place on b[].

length is the number of elements in each of the vectors.

b_shr and c_shr are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and to c.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$Re\{a_k\} \leftarrow (Re\{v_k\} \cdot Re\{c\} - Im\{v_k\} \cdot Im\{c\}) \cdot 2^{-30}$$
$$Im\{a_k\} \leftarrow (Re\{v_k\} \cdot Im\{c\} + Im\{v_k\} \cdot Re\{c\}) \cdot 2^{-30}$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 32-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$ and c is the complex 32-bit mantissa of floating-point value $c \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + c\_exp + b\_shr + c\_shr$.

The function *vect_complex_s32_mul_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Parameters

- **a** − **[out]** Complex output vector $\bar{a}$.
- **b** − **[in]** Complex input vector $\bar{b}$.
- **c_real** − **[in]** Real part of c
- **c_imag** − **[in]** Imaginary part of c
- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$.
- **b_shr** − **[in]** Right-shift appled to $\bar{b}$.
- **c_shr** − **[in]** Right-shift applied to c.

Throws ET_LOAD_STORE
Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of the output vector $\bar{a}$.

void **vect_complex_s32_set**(*complex_s32_t* a[], const int32_t b_real, const int32_t b_imag, const unsigned length)

Set each element of a complex 32-bit vector to a specified value.

a[] represents a complex 32-bit vector $\bar{a}$. a[] must begin at a word-aligned address.

b_real and b_imag are the real and imaginary parts to which each element will be set.

length is the number of elements in a[].

Operation Performed

$$a_k \leftarrow b\_real + j \cdot b\_imag$$
$$\text{for } k \in 0 \dots (length - 1)$$
$$\text{where } j^2 = -1$$

Block Floating-Point

If b is the mantissa of floating-point value $b \cdot 2^{b\_exp}$, then the output vector $\bar{a}$ are the mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp$.

Parameters

- **a** – **[out]** Complex output vector $\bar{a}$
- **b_real** – **[in]** Value to set real part of elements of $\bar{a}$ to
- **b_imag** – **[in]** Value to set imaginary part of elements of $\bar{a}$ to
- **length** – **[in]** Number of elements in $\bar{a}$

Throws **ET_LOAD_STORE**
Raised if a is not word-aligned (See *Note: Vector Alignment*)

headroom_t **vect_complex_s32_shl**(*complex_s32_t* a[], const *complex_s32_t* b[], const unsigned length, const *left_shift_t* b_shl)

Left-shift each element of a complex 32-bit vector by a specified number of bits.

a[] and b[] represent the complex 32-bit mantissa vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on b[].

length is the number of elements in $\bar{a}$ and $\bar{b}$.

b_shl is the signed arithmetic left-shift applied to each element of $\bar{b}$.

Operation Performed

$$Re\{a_k\} \leftarrow sat_{32}(\lfloor Re\{b_k\} \cdot 2^{b\_shl} \rfloor)$$
$$Im\{a_k\} \leftarrow sat_{32}(\lfloor Im\{b_k\} \cdot 2^{b\_shl} \rfloor)$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 32-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the complex 32-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $\bar{a} = \bar{b} \cdot 2^{b\_shl}$ and $a\_exp = b\_exp$.

Parameters

- **a** – **[out]** Complex output vector $\bar{a}$
- **b** – **[in]** Complex input vector $\bar{b}$
- **length** – **[in]** Number of elements in vector $\bar{b}$
- **b_shl** – **[in]** Left-shift applied to $\bar{b}$

Throws ET_LOAD_STORE

Raised if `a` or `b` is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of the output vector $\bar{a}$

---

headroom_t **vect_complex_s32_shr**(*complex_s32_t* a[], const *complex_s32_t* b[], const unsigned length, const *right_shift_t* b_shr)

Right-shift each element of a complex 32-bit vector by a specified number of bits.

`a[]` and `b[]` represent the complex 32-bit mantissa vectors $\bar{a}$ and $\bar{b}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on `b[]`.

`length` is the number of elements in $\bar{a}$ and $\bar{b}$.

`b_shr` is the signed arithmetic right-shift applied to each element of $\bar{b}$.

Operation Performed

$$Re\{a_k\} \leftarrow sat_{32}(\lfloor Re\{b_k\} \cdot 2^{-b\_shr} \rfloor)$$
$$Im\{a_k\} \leftarrow sat_{32}(\lfloor Im\{b_k\} \cdot 2^{-b\_shr} \rfloor)$$
$$\text{for } k \in 0 \ldots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 32-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the complex 32-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $\bar{a} = \bar{b} \cdot 2^{-b\_shr}$ and $a\_exp = b\_exp$.

Parameters

- **a** – **[out]** Complex output vector $\bar{a}$
- **b** – **[in]** Complex input vector $\bar{b}$
- **length** – **[in]** Number of elements in vector $\bar{b}$
- **b_shr** – **[in]** Right-shift applied to $\bar{b}$

Throws ET_LOAD_STORE

Raised if `a` or `b` is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of the output vector $\bar{a}$

---

headroom_t **vect_complex_s32_squared_mag**(int32_t a[], const *complex_s32_t* b[], const unsigned length, const *right_shift_t* b_shr)

Computes the squared magnitudes of elements of a complex 32-bit vector.

`a[]` represents the complex 32-bit mantissa vector $\bar{a}$. `b[]` represents the real 32-bit mantissa vector $\bar{b}$. Each must begin at a word-aligned address.

`length` is the number of elements in each of the vectors.

`b_shr` is the signed arithmetic right-shift appled to each element of $\bar{b}$.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$a_k \leftarrow ((Re\{b'_k\})^2 + (Im\{b'_k\})^2) \cdot 2^{-30}$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 32-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the real 32-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = 2 \cdot (b\_exp + b\_shr)$.

The function *vect_complex_s32_squared_mag_prepare()* can be used to obtain values for a_exp and b_shr based on the input exponent b_exp and headroom b_hr.

See also:

vect_complex_s32_squared_mag_prepare

Parameters

- **a** − **[out]** Complex output vector $\bar{a}$
- **b** − **[in]** Complex input vector $\bar{b}$
- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$
- **b_shr** − **[in]** Right-shift appled to $\bar{b}$

Throws ET_LOAD_STORE
Raised if `a` is not double word-aligned or `b` is not word-aligned (See *Note: Vector Alignment*)

headroom_t **vect_complex_s32_sub**(*complex_s32_t* a[], const *complex_s32_t* b[], const *complex_s32_t* c[], const unsigned length, const *right_shift_t* b_shr, const *right_shift_t* c_shr)

Subtract one complex 32-bit vector from another.

`a[]`, `b[]` and `c[]` represent the complex 32-bit mantissa vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$ respectively. Each must begin at a word-aligned address. This operation can be performed safely in-place on `b[]` or `c[]`.

`length` is the number of elements in each of the vectors.

`b_shr` and `c_shr` are the signed arithmetic right-shifts applied to each element of $\bar{b}$ and $\bar{c}$ respectively.

Operation Performed

$$b'_k \leftarrow sat_{32}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$c'_k \leftarrow sat_{32}(\lfloor c_k \cdot 2^{-c\_shr} \rfloor)$$
$$Re\{a_k\} \leftarrow Re\{b'_k\} - Re\{c'_k\}$$
$$Im\{a_k\} \leftarrow Im\{b'_k\} - Im\{c'_k\}$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ and $\bar{c}$ are the complex 32-bit mantissas of BFP vectors $\bar{b} \cdot 2^{b\_exp}$ and $\bar{c} \cdot 2^{c\_exp}$, then the resulting vector $\bar{a}$ are the complex 32-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$.

In this case, b_shr and c_shr **must** be chosen so that $a\_exp = b\_exp + b\_shr = c\_exp + c\_shr$. Adding or subtracting mantissas only makes sense if they are associated with the same exponent.

The function *vect_complex_s32_sub_prepare()* can be used to obtain values for a_exp, b_shr and c_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

See also:

vect_complex_s32_sub_prepare

Parameters

- **a** – **[out]** Complex output vector $\bar{a}$
- **b** – **[in]** Complex input vector $\bar{b}$
- **c** – **[in]** Complex input vector $\bar{c}$
- **length** – **[in]** Number of elements in vectors $\bar{a}$, $\bar{b}$ and $\bar{c}$
- **b_shr** – **[in]** Right-shift applied to $\bar{b}$
- **c_shr** – **[in]** Right-shift applied to $\bar{c}$

Throws ET_LOAD_STORE
Raised if a, b or c is not word-aligned (See *Note: Vector Alignment*)

Returns
Headroom of output vector $\bar{a}$.

void **vect_complex_s32_sum**(*complex_s64_t* *a, const *complex_s32_t* b[], const unsigned length, const *right_shift_t* b_shr)

Compute the sum of elements of a complex 32-bit vector.

a is the complex 64-bit mantissa of the resulting sum.

b[] represents the complex 32-bit mantissa vector $\bar{b}$. b[] must begin at a word-aligned address.

length is the number of elements in $\bar{b}$.

b_shr is the **unsigned** arithmetic right-shift appled to each element of $\bar{b}$. b_shr *cannot* be negative.

Operation Performed

$$b'_k \leftarrow b_k \cdot 2^{-b\_shr}$$

$$Re\{a\} \leftarrow \sum_{k=0}^{length-1} (Re\{b'_k\})$$

$$Im\{a\} \leftarrow \sum_{k=0}^{length-1} (Im\{b'_k\})$$

Block Floating-Point

If $\bar{b}$ are the mantissas of BFP vector $\bar{b} \cdot 2^{b\_exp}$, then a is the complex 64-bit mantissa of floating-point value $a \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + b\_shr$.

The function *vect_complex_s32_sum_prepare()* can be used to obtain values for a_exp and b_shr based on the input exponents b_exp and c_exp and the input headrooms b_hr and c_hr.

Additional Details

Internally the sum accumulates into four separate complex 40-bit accumulators. These accumulators apply symmetric 40-bit saturation logic (with bounds $\pm 2^{39} - 1$) with each added element. At the end, the 4 accumulators are summed together into the 64-bit fields of a. No saturation logic is applied at this final step.

In the most extreme case, each b_k may be -2^{31}. 256 of these added into the same accumulator is -2^{39} which would saturate to $-2^{39} + 1$, introducing 1 LSb of error (which may or may not be acceptable given a particular circumstance). The final result for each part then may be as large as $4 \cdot (-2^{39} + 1) = -2^{41} + 4$, each fitting into a 42-bit signed integer.

See also:

vect_complex_s32_sum_prepare

Parameters

- **a** − **[out]** Complex sum a
- **b** − **[in]** Complex input vector $\bar{b}$.
- **length** − **[in]** Number of elements in vector $\bar{b}$.
- **b_shr** − **[in]** Right-shift appled to $\bar{b}$.

Throws ET_LOAD_STORE
Raised if b is not word-aligned (See *Note: Vector Alignment*)

void **vect_complex_s32_tail_reverse**(*complex_s32_t* x[], const unsigned length)

Reverses the order of the tail of a complex 32-bit vector.

Reverses the order of elements in the tail of the complex 32-bit vector $\bar{x}$. The tail of $\bar{x}$, in this context, is all elements of $\bar{x}$ except for x_0. In other words, the first element x_0 remains where it is, and the remaining $length - 1$ elements are rearranged to have their order reversed.

This function is used when performing a forward or inverse FFT on a single sequence of real values (i.e. the mono FFT), and operates in-place on x[].

Parameter Details

x[] represents the complex 32-bit vector $\bar{x}$, which is both an input to and an output of this function. x[] must begin at a word-aligned address.

length is the number of elements in $\bar{x}$.

Operation Performed

$$x_0 \leftarrow x_0$$
$$x_k \leftarrow x_{length-k}$$
$$\text{for } k \in 1 \dots (length - 1)$$

See also:

bfp_fft_forward_mono, *bfp_fft_inverse_mono*

Parameters

- **x** − **[inout]** Complex vector to have its tail reversed.

- **length** − **[in]** Number of elements in $\bar{x}$

Throws ET_LOAD_STORE

Raised if x is not word-aligned (See *Note: Vector Alignment*)

headroom_t **vect_complex_s32_conjugate**(*complex_s32_t* a[], const *complex_s32_t* b[], const unsigned length)

Get the complex conjugate of a complex 32-bit vector.

The complex conjugate of a complex scalar $z = x + yi$ is $z^* = x - yi$. This function computes the complex conjugate of each element of $\bar{b}$ (negates the imaginary part of each element) and places the result in $\bar{a}$.

a[] is the complex 32-bit output vector $\bar{a}$.

b[] is the complex 32-bit input vector $\bar{b}$.

Both a and b must point to word-aligned addresses.

length is the number of elements in $\bar{a}$ and $\bar{b}$.

Operation Performed

$$Re\{a_k\} \leftarrow Re\{b_k\}$$
$$Im\{a_k\} \leftarrow -Im\{b_k\}$$
$$\text{for } k \in 1 \dots (length - 1)$$

Parameters

- **a** − **[out]** Complex 32-bit output vector $\bar{a}$

- **b** − **[in]** Complex 32-bit input vector $\bar{b}$

- **length** − **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

Throws ET_LOAD_STORE

Raised if a or b is not word-aligned (See *Note: Vector Alignment*)

Returns

Headroom of the output vector $\bar{a}$.

void **vect_complex_s32_to_vect_complex_s16**(int16_t a_real[], int16_t a_imag[], const *complex_s32_t* b[], const unsigned length, const *right_shift_t* b_shr)

Convert a complex 32-bit vector into a complex 16-bit vector.

This function converts a complex 32-bit mantissa vector $\bar{b}$ into a complex 16-bit mantissa vector $\bar{a}$. Conceptually, the output BFP vector $\bar{a} \cdot 2^{a\_exp}$ represents the same value as the input BFP vector $\bar{b} \cdot 2^{b\_exp}$, only with a reduced bit-depth.

In most cases b_shr should be $16 - b\_hr$, where b_hr is the headroom of the 32-bit input mantissa vector $\bar{b}$. The output exponent a_exp will then be given by

$$a\_exp = b\_exp + b\_shr$$

Parameter Details

a_real[] and a_imag[] together represent the complex 16-bit output mantissa vector $\bar{a}$, with the real part of each a_k going in a_real[] and the imaginary part going in a_imag[].

b[] represents the complex 32-bit mantissa vector $\bar{b}$.

a_real[], a_imag[] and b[] must each begin at a word-aligned address.

length is the number of elements in each of the vectors.

b_shr is the signed arithmetic right-shift applied to elements of $\bar{b}$.

Operation Performed

$$b'_k \leftarrow sat_{16}(\lfloor b_k \cdot 2^{-b\_shr} \rfloor)$$
$$Re\{a_k\} \leftarrow Re\{b'_k\}$$
$$Im\{a_k\} \leftarrow Im\{b'_k\}$$
$$\text{for } k \in 0 \dots (length - 1)$$

Block Floating-Point

If $\bar{b}$ are the complex 32-bit mantissas of a BFP vector $\bar{b} \cdot 2^{b\_exp}$, then the resulting vector $\bar{a}$ are the complex 16-bit mantissas of BFP vector $\bar{a} \cdot 2^{a\_exp}$, where $a\_exp = b\_exp + b\_shr$.

See also:

vect_s32_to_vect_s16, *vect_complex_s16_to_vect_complex_s32*

Parameters

- **a_real** – **[out]** Real part of complex output vector $\bar{a}$.

- **a_imag** – **[out]** Imaginary part of complex output vector $\bar{a}$.

- **b** – **[in]** Complex input vector $\bar{b}$.

- **length** – **[in]** Number of elements in vectors $\bar{a}$ and $\bar{b}$

- **b_shr** – **[in]** Right-shift appled to $\bar{b}$.

Throws ET_LOAD_STORE
Raised if a_real, a_imag or b are not word-aligned (See *Note: Vector Alignment*)

4.7.8 Mixed-Precision Vector API

group **vect_mixed_api**

Functions

void **mat_mul_s8_x_s16_yield_s32**(int32_t output[], const int8_t matrix[], const int16_t input_vect[], const unsigned M_rows, const unsigned N_cols, int8_t scratch[])

> Multiply an 8-bit matrix by a 16-bit vetor for a 32-bit result vector.
>
> This function multiplies an 8-bit $M \times N$ matrix $\bar{W}$ by a 16-bit N-element column vector $\bar{v}$ and returns the result as a 32-bit M-element vector $\bar{a}$.
>
> `output` is the output vector $\bar{a}$.
>
> `matrix` is the matrix $\bar{W}$.
>
> `input_vect` is the vector $\bar{v}$.
>
> `matrix` and `input_vect` must both begin at a word-aligned offsets.
>
> `M_rows` and `N_rows` are the dimensions M and N of matrix $\bar{W}$. M must be a multiple of 16, and N must be a multiple of 32.
>
> `scratch` is a pointer to a word-aligned buffer that this function may use to store intermediate results. This buffer must be at least N bytes long.
>
> The result of this multiplication is exact, so long as saturation does not occur.
>
> > **Parameters**
> >
> > - **output** – **[inout]** The output vector $\bar{a}$
> > - **matrix** – **[in]** The weight matrix $\bar{W}$
> > - **input_vect** – **[in]** The input vector $\bar{v}$
> > - **M_rows** – **[in]** The number of rows M in matrix $\bar{W}$
> > - **N_cols** – **[in]** The number of columns N in matrix $\bar{W}$
> > - **scratch** – **[in]** Scratch buffer required by this function.
> >
> > **Throws ET_LOAD_STORE**
> > Raised if `matrix` or `input_vect` is not word-aligned (See *Note: Vector Alignment*)

unsigned **vect_sXX_add_scalar**(int32_t a[], const int32_t b[], const unsigned length_bytes, const int32_t c, const int32_t d, const *right_shift_t* b_shr, const unsigned mode_bits)

> Add a scalar to a vector.
>
> Add a scalar to a vector. This works for 8, 16 or 32 bits, real or complex.
>
> `length_bytes` is the total number of bytes to be output. So, for 16-bit vectors, `length_bytes` is twice the number of elements, whereas for complex 32-bit vectors, `length_bytes` is 8 times the number of elements.
>
> `c` and `d` are the values that populate the internal buffer to be added to the input vector as follows: Internally an 8 word (32 byte) buffer is allocated (on the stack). Even-indexed words are populated with `c` and odd-indexed words are populated with `d`. For real vectors, `c` and `d` should be the same value — the reason for `d` is to allow this same function to work for complex 32-bit vectors. This also means that for 16-bit vectors, the value to be added needs to be duplicated in both the higher 2 bytes and lower 2 bytes of the word.
>
> `mode_bits` should be `0x0000` for 32-bit mode, `0x0100` for 16-bit mode or `0x0200` for 8-bit mode.

4.7.9   16-Bit Vector Prepare Functions

group `vect_s16_prepare_api`

Defines

`vect_s16_add_prepare`

> Obtain the output exponent and shifts required for a call to *vect_s16_add()*.
>
> The logic for computing the shifts and exponents of *vect_s16_add()* is identical to that for *vect_s32_add()*.
>
> This macro is provided as a convenience to developers and to make the code more readable.

> **See also:**
>
> *vect_s32_add_prepare()*

`vect_s16_add_scalar_prepare`

> Obtain the output exponent and shifts required for a call to *vect_s16_add_scalar()*.
>
> The logic for computing the shifts and exponents of *vect_s16_add_scalar()* is identical to that for *vect_s32_add()*.
>
> This macro is provided as a convenience to developers and to make the code more readable.

> **See also:**
>
> *vect_s32_add_prepare()*

`vect_s16_nmacc_prepare`

> Obtain the output exponent and shifts required for a call to *vect_s16_nmacc()*.
>
> The logic for computing the shifts and exponents of *vect_s16_nmacc()* is identical to that for *vect_s16_macc_prepare()*.
>
> This macro is provided as a convenience to developers and to make the code more readable.

> **See also:**
>
> *vect_s16_macc_prepare()*, *vect_s16_nmacc()*

`vect_s16_sub_prepare`

> Obtain the output exponent and shifts required for a call to *vect_s16_sub()*.
>
> The logic for computing the shifts and exponents of *vect_s16_sub()* is identical to that for *vect_s32_add()*.
>
> This macro is provided as a convenience to developers and to make the code more readable.

> **See also:**
>
> *vect_s32_add_prepare()*

Functions

void **vect_s16_clip_prepare**(*exponent_t* *a_exp, *right_shift_t* *b_shr, int16_t *lower_bound, int16_t *upper_bound, const *exponent_t* b_exp, const *exponent_t* bound_exp, const *headroom_t* b_hr)

Obtain the output exponent, input shift and modified bounds used by *vect_s16_clip()*.

This function is used in conjunction with *vect_s16_clip()* to bound the elements of a 32-bit BFP vector to a specified range.

This function computes `a_exp`, `b_shr`, `lower_bound` and `upper_bound`.

`a_exp` is the exponent associated with the 16-bit mantissa vector $\bar{a}$ computed by *vect_s32_clip()*.

`b_shr` is the shift parameter required by *vect_s16_clip()* to achieve the output exponent `a_exp`.

`lower_bound` and `upper_bound` are the 16-bit mantissas which indicate the lower and upper clipping bounds respectively. The values are modified by this function, and the resulting values should be passed along to *vect_s16_clip()*.

`b_exp` is the exponent associated with the input mantissa vector $\bar{b}$.

`bound_exp` is the exponent associated with the bound mantissas `lower_bound` and `upper_bound` respectively.

`b_hr` is the headroom of $\bar{b}$. If unknown, it can be obtained using *vect_s16_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

See also:

vect_s16_clip

Parameters

- **a_exp** – **[out]** Exponent associated with output mantissa vector $\bar{a}$
- **b_shr** – **[out]** Signed arithmetic right-shift for $\bar{b}$ used by *vect_s32_clip()*
- **lower_bound** – **[inout]** Lower bound of clipping range
- **upper_bound** – **[inout]** Upper bound of clipping range
- **b_exp** – **[in]** Exponent associated with input mantissa vector $\bar{b}$
- **bound_exp** – **[in]** Exponent associated with clipping bounds `lower_bound` and `upper_bound`
- **b_hr** – **[in]** Headroom of input mantissa vector $\bar{b}$

void **vect_s16_inverse_prepare**(*exponent_t* *a_exp, unsigned *scale, const int16_t b[], const *exponent_t* b_exp, const unsigned length)

Obtain the output exponent and scaling parameter used by *vect_s16_inverse()*.

This function is used in conjunction with *vect_s16_inverse()* to compute the inverse of elements of a 16-bit BFP vector.

This function computes `a_exp` and `scale`.

`a_exp` is the exponent associated with output mantissa vector $\bar{a}$, and must be chosen to avoid overflow in the smallest element of the input vector, which when inverted becomes the largest output element. To maximize precision, this function chooses `a_exp` to be the smallest exponent known to avoid saturation. The `a_exp` chosen by this function is derived from the exponent and smallest element of the input vector.

`scale` is a scaling parameter used by *vect_s16_inverse()* to achieve the chosen output exponent.

`b[]` is the input mantissa vector $\bar{b}$.

`b_exp` is the exponent associated with the input mantissa vector $\bar{b}$.

`length` is the number of elements in $\bar{b}$.

See also:

vect_s16_inverse

Parameters

- `a_exp` – **[out]** Exponent of output vector $\bar{a}$
- `scale` – **[out]** Scale factor to be applied when computing inverse
- `b` – **[in]** Input vector $\bar{b}$
- `b_exp` – **[in]** Exponent of $\bar{b}$
- `length` – **[in]** Number of elements in vector $\bar{b}$

void **vect_s16_macc_prepare**(*exponent_t* *new_acc_exp, *right_shift_t* *acc_shr, *right_shift_t* *bc_sat, const *exponent_t* acc_exp, const *exponent_t* b_exp, const *exponent_t* c_exp, const *headroom_t* acc_hr, const *headroom_t* b_hr, const *headroom_t* c_hr)

Obtain the output exponent and shifts needed by *vect_s16_macc()*.

This function is used in conjunction with *vect_s16_macc()* to perform an element-wise multiply-accumlate of 16-bit BFP vectors.

This function computes `new_acc_exp` and `acc_shr` and `bc_sat`, which are selected to maximize precision in the resulting accumulator vector without causing saturation of final or intermediate values. Normally the caller will pass these outputs to their corresponding inputs of *vect_s16_macc()*.

`acc_exp` is the exponent associated with the accumulator mantissa vector $\bar{a}$ prior to the operation, whereas `new_acc_exp` is the exponent corresponding to the updated accumulator vector.

`b_exp` and `c_exp` are the exponents associated with the complex input mantissa vectors $\bar{b}$ and $\bar{c}$ respectively.

`acc_hr`, `b_hr` and `c_hr` are the headrooms of $\bar{a}$, $\bar{b}$ and $\bar{c}$ respectively. If the headroom of any of these vectors is unknown, it can be obtained by calling *vect_s16_headroom()*. Alternatively, the value `0` can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent `desired_exp` is needed for the result (e.g. for emulating fixed-point arithmetic), the `acc_shr` and `bc_sat` produced by this function can be adjusted according to the following:

```
// Presumed to be set somewhere
exponent_t acc_exp, b_exp, c_exp;
headroom_t acc_hr, b_hr, c_hr;
exponent_t desired_exp;

...

// Call prepare
right_shift_t acc_shr, bc_sat;
vect_s16_macc_prepare(&acc_exp, &acc_shr, &bc_sat,
```

```
                          acc_exp, b_exp, c_exp,
                          acc_hr, b_hr, c_hr);

// Modify results
right_shift_t mant_shr = desired_exp - acc_exp;
acc_exp += mant_shr;
acc_shr += mant_shr;
bc_sat  += mant_shr;

// acc_shr and bc_sat may now be used in a call to vect_s16_macc()
```

When applying the above adjustment, the following conditions should be maintained:

- `bc_sat >= 0` (`bc_sat` is an *unsigned* right-shift)
- `acc_shr > -acc_hr` (Shifting any further left may cause saturation)

It is up to the user to ensure any such modification does not result in saturation or unacceptable loss of precision.

See also:

vect_s16_macc

Parameters

- **`new_acc_exp`** – **[out]** Exponent associated with output mantissa vector $\bar{a}$ (after macc)
- **`acc_shr`** – **[out]** Signed arithmetic right-shift used for $\bar{a}$ in *vect_s16_macc()*
- **`bc_sat`** – **[out]** Unsigned arithmetic right-shift applied to the product of elements b_k and c_k in *vect_s16_macc()*
- **`acc_exp`** – **[in]** Exponent associated with input mantissa vector $\bar{a}$ (before macc)
- **`b_exp`** – **[in]** Exponent associated with input mantissa vector $\bar{b}$
- **`c_exp`** – **[in]** Exponent associated with input mantissa vector $\bar{c}$
- **`acc_hr`** – **[in]** Headroom of input mantissa vector $\bar{a}$ (before macc)
- **`b_hr`** – **[in]** Headroom of input mantissa vector $\bar{b}$
- **`c_hr`** – **[in]** Headroom of input mantissa vector $\bar{c}$

void **`vect_s16_mul_prepare`**(*exponent_t* *a_exp, *right_shift_t* *a_shr, const *exponent_t* b_exp, const *exponent_t* c_exp, const *headroom_t* b_hr, const *headroom_t* c_hr)

Obtain the output exponent and output shift used by *vect_s16_mul()*.

This function is used in conjunction with *vect_s16_mul()* to perform an element-wise multiplication of two 16-bit BFP vectors.

This function computes `a_exp` and `a_shr`.

`a_exp` is the exponent associated with mantissa vector $\bar{a}$, and must be chosen to be large enough to avoid overflow when elements of $\bar{a}$ are computed. To maximize precision, this function chooses `a_exp` to be the smallest exponent known to avoid saturation (see exception below). The `a_exp` chosen by this function is derived from the exponents and headrooms of associated with the input vectors.

`a_shr` is an arithmetic right-shift applied by *vect_complex_s16_mul()* to the 32-bit products of input elements to achieve the chosen output exponent `a_exp`.

`b_exp` and `c_exp` are the exponents associated with the input mantissa vectors $\bar{b}$ and $\bar{c}$ respectively.

b_hr and c_hr are the headroom of $\bar{b}$ and $\bar{c}$ respectively. If the headroom of $\bar{b}$ or $\bar{c}$ is unknown, they can be obtained by calling *vect_s16_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent `desired_exp` is needed for the result (e.g. for emulating fixed-point arithmetic), the `a_shr` produced by this function can be adjusted according to the following:

```
exponent_t a_exp;
right_shift_t a_shr;
vect_s16_mul_prepare(&a_exp, &a_shr, b_exp, c_exp, b_hr, c_hr);
exponent_t desired_exp = ...; // Value known a priori
a_shr = a_shr + (desired_exp - a_exp);
a_exp = desired_exp;
```

When applying the above adjustment, the following conditions should be maintained:

- `a_shr >= 0`

Be aware that using a smaller value than strictly necessary for `a_shr` can result in saturation, and using larger values may result in unnecessary underflows or loss of precision.

Notes

- Using the outputs of this function, an output mantissa which would otherwise be `INT16_MIN` will instead saturate to `-INT16_MAX`. This is due to the symmetric saturation logic employed by the VPU and is a hardware feature. This is a corner case which is usually unlikely and results in 1 LSb of error when it occurs.

See also:

vect_s16_mul

Parameters

- **a_exp** – **[out]** Exponent of output elements of *vect_s16_mul()*
- **a_shr** – **[out]** Right-shift supplied to *vect_s16_mul()*
- **b_exp** – **[in]** Exponent associated with $\bar{b}$
- **c_exp** – **[in]** Exponent associated with $\bar{c}$
- **b_hr** – **[in]** Headroom of $\bar{b}$
- **c_hr** – **[in]** Headroom of $\bar{c}$

void **vect_s16_scale_prepare**(*exponent_t* *a_exp, *right_shift_t* *a_shr, const *exponent_t* b_exp, const *exponent_t* c_exp, const *headroom_t* b_hr, const *headroom_t* c_hr)

Obtain the output exponent and output shift used by *vect_s16_scale()*.

This function is used in conjunction with *vect_s16_scale()* to perform multiplication of a 16-bit BFP vector $\bar{b} \cdot 2^{b\_exp}$ by a 16-bit scalar $c \cdot 2^{c\_exp}$. The result is another 16-bit BFP vector $\bar{a} \cdot 2^{a\_exp}$.

This function computes `a_exp` and `a_shr`.

a_exp is the exponent associated with mantissa vector $\bar{a}$, and must be chosen to be large enough to avoid overflow when elements of $\bar{a}$ are computed. To maximize precision, this function chooses a_exp to be the smallest exponent known to avoid saturation (see exception below). The a_exp chosen by this function is derived from the exponents and headrooms of associated with the inputs.

a_shr is an arithmetic right-shift applied by *vect_complex_s16_scale()* to the 32-bit products of input elements to achieve the chosen output exponent a_exp.

b_exp and c_exp are the exponents associated with $\bar{b}$ and c respectively.

b_hr and c_hr are the headroom of $\bar{b}$ and c respectively. If the headroom of $\bar{b}$ or c are unknown, they can be obtained by calling *vect_s16_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent desired_exp is needed for the result (e.g. for emulating fixed-point arithmetic), the a_shr produced by this function can be adjusted according to the following:

```
exponent_t a_exp;
right_shift_t a_shr;
vect_s16_scale_prepare(&a_exp, &a_shr, b_exp, c_exp, b_hr, c_hr);
exponent_t desired_exp = ...; // Value known a priori
a_shr = a_shr + (desired_exp - a_exp);
a_exp = desired_exp;
```

When applying the above adjustment, the following conditions should be maintained:

- a_shr >= 0

Be aware that using a smaller value than strictly necessary for a_shr can result in saturation, and using larger values may result in unnecessary underflows or loss of precision.

Notes

- Using the outputs of this function, an output mantissa which would otherwise be INT16_MIN will instead saturate to -INT16_MAX. This is due to the symmetric saturation logic employed by the VPU and is a hardware feature. This is a corner case which is usually unlikely and results in 1 LSb of error when it occurs.

See also:

vect_s16_scale

Parameters

- **a_exp** – **[out]** Exponent of output elements of *vect_s16_scale()*
- **a_shr** – **[out]** Right-shift supplied to *vect_s16_scale()*
- **b_exp** – **[in]** Exponent associated with $\bar{b}$
- **c_exp** – **[in]** Exponent associated with $\bar{c}$
- **b_hr** – **[in]** Headroom of $\bar{b}$

- **c_hr** – **[in]** Headroom of $\bar{c}$

void **vect_s16_sqrt_prepare**(*exponent_t* *a_exp, *right_shift_t* *b_shr, const *exponent_t* b_exp, const *right_shift_t* b_hr)

Obtain the output exponent and shift parameter used by *vect_s16_sqrt()*.

This function is used in conjunction withx *vect_s16_sqrt()* to compute the square root of elements of a 16-bit BFP vector.

This function computes `a_exp` and `b_shr`.

`a_exp` is the exponent associated with output mantissa vector $\bar{a}$, and should be chosen to maximize the precision of the results. To that end, this function chooses `a_exp` to be the smallest exponent known to avoid saturation of the resulting mantissa vector $\bar{a}$. It is derived from the exponent and headroom of the input BFP vector.

`b_shr` is the shift parameter required by *vect_s16_sqrt()* to achieve the chosen output exponent `a_exp`.

`b_exp` is the exponent associated with the input mantissa vector $\bar{b}$.

`b_hr` is the headroom of $\bar{b}$. If it is unknown, it can be obtained using *vect_s16_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent `desired_exp` is needed for the result (e.g. for emulating fixed-point arithmetic), the `b_shr` produced by this function can be adjusted according to the following:

```
exponent_t a_exp;
right_shift_t b_shr;
vect_s16_mul_prepare(&a_exp, &b_shr, b_exp, c_exp, b_hr, c_hr);
exponent_t desired_exp = ...; // Value known a priori
b_shr = b_shr + (desired_exp - a_exp);
a_exp = desired_exp;
```

When applying the above adjustment, the following condition should be maintained:

- `b_hr + b_shr >= 0`

Be aware that using smaller values than strictly necessary for `b_shr` can result in saturation, and using larger values may result in unnecessary underflows or loss of precision.

Also, if a larger exponent is used than necessary, a larger `depth` parameter (see *vect_s16_sqrt()*) will be required to achieve the same precision, as the results are computed bit by bit, starting with the most significant bit.

See also:

vect_s16_sqrt

Parameters

- **a_exp** – **[out]** Exponent of outputs of *vect_s16_sqrt()*
- **b_shr** – **[out]** Right-shift to be applied to elements of $\bar{b}$
- **b_exp** – **[in]** Exponent of vector{b}
- **b_hr** – **[in]** Headroom of vector{b}

4.7.10　32-Bit Vector Prepare Functions

group `vect_s32_prepare_api`

Defines

`vect_s32_add_scalar_prepare`

> Obtain the output exponent and shifts required for a call to `vect_s32_add_scalar()`.
>
> The logic for computing the shifts and exponents of `vect_s32_add_scalar()` is identical to that for `vect_s32_add()`.
>
> This macro is provided as a convenience to developers and to make the code more readable.
>
> **See also:**
>
> *vect_s32_add_prepare()*

`vect_s32_nmacc_prepare`

> Obtain the output exponent and shifts required for a call to *vect_s32_nmacc()*.
>
> The logic for computing the shifts and exponents of `vect_s32_nmacc()` is identical to that for `vect_s32_macc_prepare()`.
>
> This macro is provided as a convenience to developers and to make the code more readable.
>
> **See also:**
>
> *vect_s32_macc_prepare()*, *vect_s32_nmacc()*

`vect_s32_scale_prepare`

> Obtain the output exponent and shifts required for a call to `vect_s32_scale()`.
>
> The logic for computing the shifts and exponents of `vect_s32_scale()` is identical to that for `vect_s32_mul()`.
>
> This macro is provided as a convenience to developers and to make the code more readable.
>
> **See also:**
>
> *vect_s32_mul_prepare()*

`vect_s32_sub_prepare`

> Obtain the output exponent and shifts required for a call to `vect_s32_sub()`.
>
> The logic for computing the shifts and exponents of `vect_s32_sub()` is identical to that for `vect_s32_add()`.
>
> This macro is provided as a convenience to developers and to make the code more readable.
>
> **See also:**
>
> *vect_s32_add_prepare()*

Functions

void **vect_s32_add_prepare**(*exponent_t* *a_exp, *right_shift_t* *b_shr, *right_shift_t* *c_shr, const *exponent_t* b_exp, const *exponent_t* c_exp, const *headroom_t* b_hr, const *headroom_t* c_hr)

Obtain the output exponent and input shifts to add or subtract two 16- or 32-bit BFP vectors.

The block floating-point functions in this library which add or subtract vectors are of the general form:

$$\bar{a} \cdot 2^{a\_exp} = \bar{b} \cdot 2^{b\_exp} \pm \bar{c} \cdot 2^{c\_exp} \}$$

$\bar{b}$ and $\bar{c}$ are the input mantissa vectors with exponents b_exp and c_exp, which are shared by each element of their respective vectors. $\bar{a}$ is the output mantissa vector with exponent a_exp. Two additional properties, b_hr and c_hr, which are the headroom of mantissa vectors $\bar{b}$ and $\bar{c}$ respectively, are required by this function.

In order to avoid any overflows in the output mantissas, the output exponent a_exp must be chosen such that the largest (in the sense of absolute value) possible output mantissa will fit into the allotted space (e.g. 32 bits for *vect_s32_add()*). Once a_exp is chosen, the input bit-shifts b_shr and c_shr are calculated to achieve that resulting exponent.

This function chooses a_exp to be the minimum exponent known to avoid overflows, given the input exponents (b_exp and c_exp) and input headroom (b_hr and c_hr).

This function is used calculate the output exponent and input bit-shifts for each of the following functions:

- *vect_s16_add()*
- *vect_s32_add()*
- *vect_s16_sub()*
- *vect_s32_sub()*
- *vect_complex_s16_add()*
- *vect_complex_s32_add()*
- *vect_complex_s16_sub()*
- *vect_complex_s32_sub()*

Adjusting Output Exponents

If a specific output exponent `desired_exp` is needed for the result (e.g. for emulating fixed-point arithmetic), the `b_shr` and `c_shr` produced by this function can be adjusted according to the following:

```
exponent_t desired_exp = ...; // Value known a priori
right_shift_t new_b_shr = b_shr + (desired_exp - a_exp);
right_shift_t new_c_shr = c_shr + (desired_exp - a_exp);
```

When applying the above adjustment, the following conditions should be maintained:

- `b_hr + b_shr >= 0`
- `c_hr + c_shr >= 0`

Be aware that using smaller values than strictly necessary for `b_shr` and `c_shr` can result in saturation, and using larger values may result in unnecessary underflows or loss of precision.

Notes

- If b_hr or c_hr are unknown, they can be calculated using the appropriate headroom function (e.g. *vect_complex_s16_headroom()* for complex 16-bit vectors) or the value 0 can always be safely used (but may result in reduced precision).

See also:

vect_s16_add, *vect_s32_add*, *vect_s16_sub*, *vect_s32_sub*, *vect_complex_s16_add*, *vect_complex_s32_add*, *vect_complex_s16_sub*, *vect_complex_s32_sub*

Parameters

- **a_exp** − **[out]** Output exponent associated with output mantissa vector $\bar{a}$
- **b_shr** − **[out]** Signed arithmetic right-shift to be applied to elements of $\bar{b}$. Used by the function which computes the output mantissas $\bar{a}$
- **c_shr** − **[out]** Signed arithmetic right-shift to be applied to elements of $\bar{c}$. Used by the function which computes the output mantissas $\bar{a}$
- **b_exp** − **[in]** Exponent of BFP vector $\bar{b}$
- **c_exp** − **[in]** Exponent of BFP vector $\bar{c}$
- **b_hr** − **[in]** Headroom of BFP vector $\bar{b}$
- **c_hr** − **[in]** Headroom of BFP vector $\bar{c}$

void **vect_s32_clip_prepare**(*exponent_t* *a_exp, *right_shift_t* *b_shr, int32_t *lower_bound, int32_t *upper_bound, const *exponent_t* b_exp, const *exponent_t* bound_exp, const *headroom_t* b_hr)

Obtain the output exponent, input shift and modified bounds used by *vect_s32_clip()*.

This function is used in conjunction with *vect_s32_clip()* to bound the elements of a 32-bit BFP vector to a specified range.

This function computes a_exp, b_shr, lower_bound and upper_bound.

a_exp is the exponent associated with the 32-bit mantissa vector $\bar{a}$ computed by *vect_s32_clip()*.

b_shr is the shift parameter required by *vect_s32_clip()* to achieve the output exponent a_exp.

lower_bound and upper_bound are the 32-bit mantissas which indicate the lower and upper clipping bounds respectively. The values are modified by this function, and the resulting values should be passed along to *vect_s32_clip()*.

b_exp is the exponent associated with the input mantissa vector $\bar{b}$.

bound_exp is the exponent associated with the bound mantissas lower_bound and upper_bound respectively.

b_hr is the headroom of $\bar{b}$. If unknown, it can be obtained using *vect_s32_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

See also:

vect_s32_clip

Parameters

- **a_exp** − **[out]** Exponent associated with output mantissa vector $\bar{a}$

- **b_shr** – **[out]** Signed arithmetic right-shift for $\bar{b}$ used by *vect_s32_clip()*

- **lower_bound** – **[inout]** Lower bound of clipping range

- **upper_bound** – **[inout]** Upper bound of clipping range

- **b_exp** – **[in]** Exponent associated with input mantissa vector $\bar{b}$

- **bound_exp** – **[in]** Exponent associated with clipping bounds lower_bound and upper_bound

- **b_hr** – **[in]** Headroom of input mantissa vector $\bar{b}$

void **vect_s32_dot_prepare**(*exponent_t* *a_exp, *right_shift_t* *b_shr, *right_shift_t* *c_shr, const *exponent_t* b_exp, const *exponent_t* c_exp, const *headroom_t* b_hr, const *headroom_t* c_hr, const unsigned length)

Obtain the output exponent and input shift used by *vect_s32_dot()*.

This function is used in conjunction with *vect_s32_dot()* to compute the inner product of two 32-bit BFP vectors.

This function computes a_exp, b_shr and c_shr.

a_exp is the exponent associated with the 64-bit mantissa a returned by *vect_s32_dot()*, and must be chosen to be large enough to avoid saturation when a is computed. To maximize precision, this function chooses a_exp to be the smallest exponent known to avoid saturation (see exception below). The a_exp chosen by this function is derived from the exponents and headrooms associated with the input vectors.

b_shr and c_shr are the shift parameters required by *vect_s32_dot()* to achieve the chosen output exponent a_exp.

b_exp and c_exp are the exponents associated with the input mantissa vectors $\bar{b}$ and $\bar{c}$ respectively.

b_hr and c_hr are the headroom of $\bar{b}$ and $\bar{c}$ respectively. If either is unknown, they can be obtained using *vect_s32_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

length is the number of elements in the input mantissa vectors $\bar{b}$ and $\bar{c}$.

Adjusting Output Exponents

If a specific output exponent desired_exp is needed for the result (e.g. for emulating fixed-point arithmetic), the b_shr and c_shr produced by this function can be adjusted according to the following:

```
exponent_t desired_exp = ...; // Value known a priori
right_shift_t new_b_shr = b_shr + (desired_exp - a_exp);
right_shift_t new_c_shr = c_shr + (desired_exp - a_exp);
```

When applying the above adjustment, the following conditions should be maintained:

- b_hr + b_shr >= 0

- c_hr + c_shr >= 0

Be aware that using smaller values than strictly necessary for b_shr or c_shr can result in saturation, and using larger values may result in unnecessary underflows or loss of precision.

See also:

vect_s32_dot

Parameters

- **a_exp** – **[out]** Exponent associated with output mantissa a
- **b_shr** – **[out]** Signed arithmetic right-shift for $\bar{b}$ used by *vect_s32_dot()*
- **c_shr** – **[out]** Signed arithmetic right-shift for $\bar{c}$ used by *vect_s32_dot()*
- **b_exp** – **[in]** Exponent associated with input mantissa vector $\bar{b}$
- **c_exp** – **[in]** Exponent associated with input mantissa vector $\bar{b}$
- **b_hr** – **[in]** Headroom of input mantissa vector $\bar{b}$
- **c_hr** – **[in]** Headroom of input mantissa vector $\bar{b}$
- **length** – **[in]** Number of elements in vectors $\bar{b}$ and $\bar{c}$

void **vect_s32_energy_prepare**(*exponent_t* *a_exp, *right_shift_t* *b_shr, const unsigned length, const *exponent_t* b_exp, const *headroom_t* b_hr)

Obtain the output exponent and input shift used by *vect_s32_energy()*.

This function is used in conjunction with *vect_s32_energy()* to compute the inner product of a 32-bit BFP vector with itself.

This function computes `a_exp` and `b_shr`.

`a_exp` is the exponent associated with the 64-bit mantissa a returned by *vect_s32_energy()*, and must be chosen to be large enough to avoid saturation when a is computed. To maximize precision, this function chooses `a_exp` to be the smallest exponent known to avoid saturation (see exception below). The `a_exp` chosen by this function is derived from the exponent and headroom associated with the input vector.

`b_shr` is the shift parameter required by *vect_s32_energy()* to achieve the chosen output exponent `a_exp`.

`b_exp` is the exponent associated with the input mantissa vector $\bar{b}$.

`b_hr` is the headroom of $\bar{b}$. If it is unknown, it can be obtained using *vect_s32_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

`length` is the number of elements in the input mantissa vector $\bar{b}$.

Adjusting Output Exponents

If a specific output exponent `desired_exp` is needed for the result (e.g. for emulating fixed-point arithmetic), the `b_shr` produced by this function can be adjusted according to the following:

```
exponent_t desired_exp = ...; // Value known a priori
right_shift_t new_b_shr = b_shr + (desired_exp - a_exp);
```

When applying the above adjustment, the following condition should be maintained:

- `b_hr + b_shr >= 0`

Be aware that using smaller values than strictly necessary for `b_shr` can result in saturation, and using larger values may result in unnecessary underflows or loss of precision.

See also:

vect_s32_energy

Parameters

- **a_exp** – **[out]** Exponent of outputs of *vect_s32_energy()*

- **b_shr** – **[out]** Right-shift to be applied to elements of $\bar{b}$

- **length** – **[in]** Number of elements in vector $\bar{b}$

- **b_exp** – **[in]** Exponent of vector{b}

- **b_hr** – **[in]** Headroom of vector{b}

void **vect_s32_inverse_prepare**(*exponent_t* *a_exp, unsigned *scale, const int32_t b[], const *exponent_t* b_exp, const unsigned length)

Obtain the output exponent and scale used by *vect_s32_inverse()*.

This function is used in conjunction with *vect_s32_inverse()* to compute the inverse of elements of a 32-bit BFP vector.

This function computes a_exp and scale.

a_exp is the exponent associated with output mantissa vector $\bar{a}$, and must be chosen to avoid overflow in the smallest element of the input vector, which when inverted becomes the largest output element. To maximize precision, this function chooses a_exp to be the smallest exponent known to avoid saturation. The a_exp chosen by this function is derived from the exponent and smallest element of the input vector.

scale is a scaling parameter used by *vect_s32_inverse()* to achieve the chosen output exponent.

b[] is the input mantissa vector $\bar{b}$.

b_exp is the exponent associated with the input mantissa vector $\bar{b}$.

length is the number of elements in $\bar{b}$.

See also:

vect_s32_inverse

Parameters

- **a_exp** – **[out]** Exponent of output vector $\bar{a}$

- **scale** – **[out]** Scale factor to be applied when computing inverse

- **b** – **[in]** Input vector $\bar{b}$

- **b_exp** – **[in]** Exponent of $\bar{b}$

- **length** – **[in]** Number of elements in vector $\bar{b}$

void **vect_s32_macc_prepare**(*exponent_t* *new_acc_exp, *right_shift_t* *acc_shr, *right_shift_t* *b_shr, *right_shift_t* *c_shr, const *exponent_t* acc_exp, const *exponent_t* b_exp, const *exponent_t* c_exp, const *headroom_t* acc_hr, const *headroom_t* b_hr, const *headroom_t* c_hr)

Obtain the output exponent and shifts needed by *vect_s32_macc()*.

This function is used in conjunction with *vect_s32_macc()* to perform an element-wise multiply-accumlate of 32-bit BFP vectors.

This function computes new_acc_exp, acc_shr, b_shr and c_shr, which are selected to maximize precision in the resulting accumulator vector without causing saturation of final or intermediate values. Normally the caller will pass these outputs to their corresponding inputs of *vect_s32_macc()*.

acc_exp is the exponent associated with the accumulator mantissa vector $\bar{a}$ prior to the operation, whereas new_acc_exp is the exponent corresponding to the updated accumulator vector.

b_exp and c_exp are the exponents associated with the complex input mantissa vectors $\bar{b}$ and $\bar{c}$ respectively.

acc_hr, b_hr and c_hr are the headrooms of $\bar{a}$, $\bar{b}$ and $\bar{c}$ respectively. If the headroom of any of these vectors is unknown, it can be obtained by calling *vect_s32_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent `desired_exp` is needed for the result (e.g. for emulating fixed-point arithmetic), the `acc_shr` and `bc_sat` produced by this function can be adjusted according to the following:

```
// Presumed to be set somewhere
exponent_t acc_exp, b_exp, c_exp;
headroom_t acc_hr, b_hr, c_hr;
exponent_t desired_exp;

...

// Call prepare
right_shift_t acc_shr, b_shr, c_shr;
vect_s32_macc_prepare(&acc_exp, &acc_shr, &b_shr, &c_shr,
                      acc_exp, b_exp, c_exp,
                      acc_hr, b_hr, c_hr);

// Modify results
right_shift_t mant_shr = desired_exp - acc_exp;
acc_exp += mant_shr;
acc_shr += mant_shr;
b_shr  += mant_shr;
c_shr  += mant_shr;

// acc_shr, b_shr and c_shr may now be used in a call to vect_s32_macc()
```

When applying the above adjustment, the following conditions should be maintained:

- `acc_shr > -acc_hr` (Shifting any further left may cause saturation)
- `b_shr => -b_hr` (Shifting any further left may cause saturation)
- `c_shr => -c_hr` (Shifting any further left may cause saturation)

It is up to the user to ensure any such modification does not result in saturation or unacceptable loss of precision.

See also:

vect_s32_macc

Parameters

- **new_acc_exp** – **[out]** Exponent associated with output mantissa vector $\bar{a}$ (after macc)

- **acc_shr** – **[out]** Signed arithmetic right-shift used for $\bar{a}$ in *vect_s32_macc()*

- **b_shr** – **[out]** Signed arithmetic right-shift used for $\bar{b}$ in *vect_s32_macc()*

- **c_shr** – **[out]** Signed arithmetic right-shift used for $\bar{c}$ in *vect_s32_macc()*

- **acc_exp** – **[in]** Exponent associated with input mantissa vector $\bar{a}$ (before macc)

- **b_exp** – **[in]** Exponent associated with input mantissa vector $\bar{b}$

- **c_exp** – **[in]** Exponent associated with input mantissa vector $\bar{c}$

- **acc_hr** – **[in]** Headroom of input mantissa vector $\bar{a}$ (before macc)

- **b_hr** – **[in]** Headroom of input mantissa vector $\bar{b}$

- **c_hr** – **[in]** Headroom of input mantissa vector $\bar{c}$

void **vect_s32_mul_prepare**(*exponent_t* *a_exp, *right_shift_t* *b_shr, *right_shift_t* *c_shr, const *exponent_t* b_exp, const *exponent_t* c_exp, const *headroom_t* b_hr, const *headroom_t* c_hr)

Obtain the output exponent and input shifts used by *vect_s32_mul()*.

This function is used in conjunction with *vect_s32_mul()* to perform an element-wise multiplication of two 32-bit BFP vectors.

This function computes a_exp, b_shr, c_shr.

a_exp is the exponent associated with mantissa vector $\bar{a}$, and must be chosen to be large enough to avoid overflow when elements of $\bar{a}$ are computed. To maximize precision, this function chooses a_exp to be the smallest exponent known to avoid saturation (see exception below). The a_exp chosen by this function is derived from the exponents and headrooms of associated with the input vectors.

b_shr and c_shr are the shift parameters required by *vect_complex_s32_mul()* to achieve the chosen output exponent a_exp.

b_exp and c_exp are the exponents associated with the input mantissa vectors $\bar{b}$ and $\bar{c}$ respectively.

b_hr and c_hr are the headroom of $\bar{b}$ and $\bar{c}$ respectively. If the headroom of $\bar{b}$ or $\bar{c}$ is unknown, they can be obtained by calling *vect_s32_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent desired_exp is needed for the result (e.g. for emulating fixed-point arithmetic), the b_shr and c_shr produced by this function can be adjusted according to the following:

```
exponent_t desired_exp = ...; // Value known a priori
right_shift_t new_b_shr = b_shr + (desired_exp - a_exp);
right_shift_t new_c_shr = c_shr + (desired_exp - a_exp);
```

When applying the above adjustment, the following conditions should be maintained:

- b_hr + b_shr >= 0

- c_hr + c_shr >= 0

Be aware that using smaller values than strictly necessary for b_shr and c_shr can result in saturation, and using larger values may result in unnecessary underflows or loss of precision.

Notes

- Using the outputs of this function, an output mantissa which would otherwise be INT32_MIN will instead saturate to -INT32_MAX. This is due to the symmetric saturation logic employed by the VPU and is a hardware feature. This is a corner case which is usually unlikely and results in 1 LSb of error when it occurs.

Parameters

- **a_exp** – **[out]** Exponent of output elements of *vect_s32_mul()*
- **b_shr** – **[out]** Right-shift to be applied to elements of $\bar{b}$
- **c_shr** – **[out]** Right-shift to be applied to elemetns of $\bar{c}$
- **b_exp** – **[in]** Exponent of $\bar{b}$
- **c_exp** – **[in]** Exponent of $\bar{c}$
- **b_hr** – **[in]** Headroom of $\bar{b}$
- **c_hr** – **[in]** Headroom of $\bar{c}$

void **vect_s32_sqrt_prepare**(*exponent_t* *a_exp, *right_shift_t* *b_shr, const *exponent_t* b_exp, const *right_shift_t* b_hr)

Obtain the output exponent and shift parameter used by *vect_s32_sqrt()*.

This function is used in conjunction withx *vect_s32_sqrt()* to compute the square root of elements of a 32-bit BFP vector.

This function computes a_exp and b_shr.

a_exp is the exponent associated with output mantissa vector $\bar{a}$, and should be chosen to maximize the precision of the results. To that end, this function chooses a_exp to be the smallest exponent known to avoid saturation of the resulting mantissa vector $\bar{a}$. It is derived from the exponent and headroom of the input BFP vector.

b_shr is the shift parameter required by *vect_s32_sqrt()* to achieve the chosen output exponent a_exp.

b_exp is the exponent associated with the input mantissa vector $\bar{b}$.

b_hr is the headroom of $\bar{b}$. If it is unknown, it can be obtained using *vect_s32_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent desired_exp is needed for the result (e.g. for emulating fixed-point arithmetic), the b_shr produced by this function can be adjusted according to the following:

```
exponent_t a_exp;
right_shift_t b_shr;
vect_s16_mul_prepare(&a_exp, &b_shr, b_exp, c_exp, b_hr, c_hr);
exponent_t desired_exp = ...; // Value known a priori
b_shr = b_shr + (desired_exp - a_exp);
a_exp = desired_exp;
```

When applying the above adjustment, the following condition should be maintained:

- b_hr + b_shr >= 0

Be aware that using smaller values than strictly necessary for b_shr can result in saturation, and using larger values may result in unnecessary underflows or loss of precision.

Also, if a larger exponent is used than necessary, a larger `depth` parameter (see *vect_s32_sqrt()*) will be required to achieve the same precision, as the results are computed bit by bit, starting with the most significant bit.

See also:

vect_s32_sqrt

Parameters

- **a_exp** – **[out]** Exponent of outputs of *vect_s32_sqrt()*
- **b_shr** – **[out]** Right-shift to be applied to elements of $\bar{b}$
- **b_exp** – **[in]** Exponent of vector{b}
- **b_hr** – **[in]** Headroom of vector{b}

void **vect_2vec_prepare**(*exponent_t* *a_exp, *right_shift_t* *b_shr, *right_shift_t* *c_shr, const *exponent_t* b_exp, const *exponent_t* c_exp, const *headroom_t* b_hr, const *headroom_t* c_hr, const *headroom_t* extra_operand_hr)

Obtain the output exponent and input shifts required to perform a binary add-like operation.

This function computes the output exponent and input shifts required for BFP operations which take two vectors as input, where the operation is "add-like".

Here, "add-like" operations are loosely defined as those which require input vectors to share an exponent before their mantissas can be meaningfully used to perform that operation.

For example, consider adding $3 \cdot 2^x + 4 \cdot 2^y$. If $x = y$, then the mantissas can be added directly to get a meaningful result $(3 + 4) \cdot 2^x$. If $x \neq y$ however, adding the mantissas together is meaningless. Before the mantissas can be added in this case, one or both of the input mantissas must be shifted so that the representations correspond to the same exponent. Likewise, similar logic applies to binary comparisons.

This is in contrast to a "multiply-like" operation, which does not have this same requirement (e.g. $a \cdot 2^x \cdot b \cdot 2^y = ab \cdot 2^{x+y}$, regardless of whether $x = y$).

For a general operation like:

$$\bar{a} \cdot 2^{a\_exp} = \bar{b} \cdot 2^{b\_exp} \oplus \bar{c} \cdot 2^{c\_exp}$$

$\bar{b}$ and $\bar{c}$ are the input mantissa vectors with exponents b_exp and c_exp, which are shared by each element of their respective vectors. $\bar{a}$ is the output mantissa vector with exponent a_exp. Two additional properties, b_hr and c_hr, which are the headroom of mantissa vectors $\bar{b}$ and $\bar{c}$ respectively, are required by this function.

In addition to a_exp, this function computes b_shr and c_shr, signed arithmetic right-shifts applied to the mantissa vectors $\bar{b}$ and $\bar{c}$ so that the add-like $\oplus$ operation can be applied.

This function chooses a_exp to be the minimum exponent which can be used to express both $\bar{B}$ and $\bar{C}$ without saturation of their mantissas, and which leaves both $\bar{b}$ and $\bar{c}$ with at least `extra_operand_hr` bits of headroom. The shifts b_shr and c_shr are derived from a_exp using b_exp and c_exp.

Adjusting Output Exponents

If a specific output exponent `desired_exp` is needed for the result (e.g. for emulating fixed-point arithmetic), the `b_shr` and `c_shr` produced by this function can be adjusted according to the following:

```
exponent_t desired_exp = ...; // Value known a priori
right_shift_t new_b_shr = b_shr + (desired_exp - a_exp);
right_shift_t new_c_shr = c_shr + (desired_exp - a_exp);
```

When applying the above adjustment, the following conditions should be maintained:

- `b_hr + b_shr >= 0`

- `c_hr + c_shr >= 0`

Be aware that using smaller values than strictly necessary for `b_shr` and `c_shr` can result in saturation, and using larger values may result in unnecessary underflows or loss of precision.

Notes

- If b_hr or c_hr are unknown, they can be calculated using the appropriate headroom function (e.g. *vect_complex_s16_headroom()* for complex 16-bit vectors) or the value `0` can always be safely used (but may result in reduced precision).

Parameters

- `a_exp` – **[out]** Output exponent associated with output mantissa vector $\bar{a}$

- `b_shr` – **[out]** Signed arithmetic right-shift to be applied to elements of $\bar{b}$. Used by the function which computes the output mantissas $\bar{a}$

- `c_shr` – **[out]** Signed arithmetic right-shift to be applied to elements of $\bar{c}$. Used by the function which computes the output mantissas $\bar{a}$

- `b_exp` – **[in]** Exponent of BFP vector $\bar{b}$

- `c_exp` – **[in]** Exponent of BFP vector $\bar{c}$

- `b_hr` – **[in]** Headroom of BFP vector $\bar{b}$

- `c_hr` – **[in]** Headroom of BFP vector $\bar{c}$

- `extra_operand_hr` – **[in]** The minimum amount of headroom that will be left in the mantissa vectors following the arithmetic right-shift, as required by some operations.

4.7.11 16-Bit Complex Vector Prepare Functions

group **vect_complex_s16_prepare_api**

Defines

vect_complex_s16_add_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s16_add()*.

The logic for computing the shifts and exponents of *vect_complex_s16_add()* is identical to that for *vect_s32_add()*.

This macro is provided as a convenience to developers and to make the code more readable.

See also:

vect_s32_add_prepare()

vect_complex_s16_add_scalar_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s16_add_scalar()*.

The logic for computing the shifts and exponents of *vect_complex_s16_add_scalar()* is identical to that for *vect_s32_add()*.

This macro is provided as a convenience to developers and to make the code more readable.

See also:

vect_s16_add_prepare()

vect_complex_s16_conj_mul_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s16_conj_mul()*.

The logic for computing the shifts and exponents of *vect_complex_s16_conj_mul()* is identical to that for *vect_complex_s16_mul()*.

This macro is provided as a convenience to developers and to make the code more readable.

See also:

vect_complex_s16_mul_prepare()

vect_complex_s16_nmacc_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s16_nmacc()*.

The logic for computing the shifts and exponents of *vect_complex_s16_nmacc()* is identical to that for *vect_complex_s16_macc()*.

This macro is provided as a convenience to developers and to make the code more readable.

See also:

vect_complex_s16_macc_prepare(), *vect_complex_s16_nmacc()*

vect_complex_s16_conj_macc_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s16_conj_macc()*.

The logic for computing the shifts and exponents of *vect_complex_s16_conj_macc()* is identical to that for *vect_complex_s16_macc()*.

This macro is provided as a convenience to developers and to make the code more readable.

See also:

vect_complex_s16_macc_prepare(), *vect_complex_s16_conj_macc()*

vect_complex_s16_conj_nmacc_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s16_conj_nmacc()*.

The logic for computing the shifts and exponents of `vect_complex_s16_conj_nmacc()` is identical to that for `vect_complex_s16_macc()`.

This macro is provided as a convenience to developers and to make the code more readable.

See also:

vect_complex_s16_macc_prepare(), *vect_complex_s16_conj_nmacc()*

vect_complex_s16_mag_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s16_mag()*.

The logic for computing the shifts and exponents of `vect_complex_s16_mag()` is identical to that for `vect_complex_s32_mag()`.

This macro is provided as a convenience to developers and to make the code more readable.

See also:

vect_complex_s32_mag_prepare()

vect_complex_s16_real_scale_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s16_real_scale()*.

The logic for computing the shifts and exponents of `vect_complex_s16_real_scale()` is identical to that for `vect_s32_scale()`.

This macro is provided as a convenience to developers and to make the code more readable.

See also:

vect_s16_scale_prepare()

vect_complex_s16_scale_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s16_scale()*.

The logic for computing the shifts and exponents of `vect_complex_s16_scale()` is identical to that for `vect_complex_s32_mul()`.

This macro is provided as a convenience to developers and to make the code more readable.

See also:

vect_complex_s16_mul_prepare()

vect_complex_s16_sub_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s16_sub()*.

The logic for computing the shifts and exponents of `vect_complex_s16_sub()` is identical to that for `vect_s32_add()`.

This macro is provided as a convenience to developers and to make the code more readable.

Functions

void **vect_complex_s16_macc_prepare**(*exponent_t* *new_acc_exp, *right_shift_t* *acc_shr, *right_shift_t* *bc_sat, const *exponent_t* acc_exp, const *exponent_t* b_exp, const *exponent_t* c_exp, const *headroom_t* acc_hr, const *headroom_t* b_hr, const *headroom_t* c_hr)

Obtain the output exponent and shifts needed by *vect_complex_s16_macc()*.

This function is used in conjunction with *vect_complex_s16_macc()* to perform an element-wise multiply-accumlate of complex 16-bit BFP vectors.

This function computes `new_acc_exp` and `acc_shr` and `bc_sat`, which are selected to maximize precision in the resulting accumulator vector without causing saturation of final or intermediate values. Normally the caller will pass these outputs to their corresponding inputs of *vect_complex_s16_macc()*.

`acc_exp` is the exponent associated with the accumulator mantissa vector $\bar{a}$ prior to the operation, whereas `new_acc_exp` is the exponent corresponding to the updated accumulator vector.

`b_exp` and `c_exp` are the exponents associated with the complex input mantissa vectors $\bar{b}$ and $\bar{c}$ respectively.

`acc_hr`, `b_hr` and `c_hr` are the headrooms of $\bar{a}, \bar{b}$ and $\bar{c}$ respectively. If the headroom of any of these vectors is unknown, it can be obtained by calling *vect_complex_s16_headroom()*. Alternatively, the value `0` can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent `desired_exp` is needed for the result (e.g. for emulating fixed-point arithmetic), the `acc_shr` and `bc_sat` produced by this function can be adjusted according to the following:

```
// Presumed to be set somewhere
exponent_t acc_exp, b_exp, c_exp;
headroom_t acc_hr, b_hr, c_hr;
exponent_t desired_exp;

...

// Call prepare
right_shift_t acc_shr, bc_sat;
vect_complex_s16_macc_prepare(&acc_exp, &acc_shr, &bc_sat,
                              acc_exp, b_exp, c_exp,
                              acc_hr, b_hr, c_hr);

// Modify results
right_shift_t mant_shr = desired_exp - acc_exp;
acc_exp += mant_shr;
acc_shr += mant_shr;
bc_sat  += mant_shr;

// acc_shr and bc_sat may now be used in a call to vect_complex_s16_macc()
```

When applying the above adjustment, the following conditions should be maintained:

- `bc_sat >= 0` (`bc_sat` is an *unsigned* right-shift)
- `acc_shr > -acc_hr` (Shifting any further left may cause saturation)

It is up to the user to ensure any such modification does not result in saturation or unacceptable loss of precision.

See also:

vect_complex_s16_macc

Parameters

- `new_acc_exp` − **[out]** Exponent associated with output mantissa vector $\bar{a}$ (after macc)

- `acc_shr` − **[out]** Signed arithmetic right-shift used for $\bar{a}$ in *vect_complex_s16_macc()*

- `bc_sat` − **[out]** Unsigned arithmetic right-shift applied to the product of elements b_k and c_k in *vect_complex_s16_macc()*

- `acc_exp` − **[in]** Exponent associated with input mantissa vector $\bar{a}$ (before macc)

- `b_exp` − **[in]** Exponent associated with input mantissa vector $\bar{b}$

- `c_exp` − **[in]** Exponent associated with input mantissa vector $\bar{c}$

- `acc_hr` − **[in]** Headroom of input mantissa vector $\bar{a}$ (before macc)

- `b_hr` − **[in]** Headroom of input mantissa vector $\bar{b}$

- `c_hr` − **[in]** Headroom of input mantissa vector $\bar{c}$

void **vect_complex_s16_mul_prepare**(*exponent_t* *a_exp, *right_shift_t* *a_shr, const *exponent_t* b_exp, const *exponent_t* c_exp, const *headroom_t* b_hr, const *headroom_t* c_hr)

Obtain the output exponent and output shift used by *vect_complex_s16_mul()* and *vect_complex_s16_conj_mul()*.

This function is used in conjunction with *vect_complex_s16_mul()* to perform a complex element-wise multiplication of two complex 16-bit BFP vectors.

This function computes `a_exp` and `a_shr`.

`a_exp` is the exponent associated with mantissa vector $\bar{a}$, and must be chosen to be large enough to avoid overflow when elements of $\bar{a}$ are computed. To maximize precision, this function chooses `a_exp` to be the smallest exponent known to avoid saturation (see exception below). The `a_exp` chosen by this function is derived from the exponents and headrooms of associated with the input vectors.

`a_shr` is the shift parameter required by *vect_complex_s16_mul()* to achieve the chosen output exponent `a_exp`.

`b_exp` and `c_exp` are the exponents associated with the input mantissa vectors $\bar{b}$ and $\bar{c}$ respectively.

`b_hr` and `c_hr` are the headroom of $\bar{b}$ and $\bar{c}$ respectively. If the headroom of $\bar{b}$ or $\bar{c}$ is unknown, they can be obtained by calling *vect_complex_s16_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent `desired_exp` is needed for the result (e.g. for emulating fixed-point arithmetic), the `a_shr` and `c_shr` produced by this function can be adjusted according to the following:

```
exponent_t desired_exp = ...; // Value known a priori
right_shift_t new_a_shr = a_shr + (desired_exp - a_exp);
```

When applying the above adjustment, the following conditions should be maintained:

- `new_a_shr >= 0`

Be aware that using smaller values than strictly necessary for `a_shr` can result in saturation, and using larger values may result in unnecessary underflows or loss of precision.

Notes

- Using the outputs of this function, an output mantissa which would otherwise be `INT16_MIN` will instead saturate to `-INT16_MAX`. This is due to the symmetric saturation logic employed by the VPU and is a hardware feature. This is a corner case which is usually unlikely and results in 1 LSb of error when it occurs.

See also:

vect_complex_s16_conj_mul, *vect_complex_s16_mul*

Parameters

- **a_exp** – **[out]** Exponent associated with output mantissa vector $\bar{a}$
- **a_shr** – **[out]** Unsigned arithmetic right-shift for $\bar{b}$ used by *vect_complex_s16_mul()*
- **b_exp** – **[in]** Exponent associated with input mantissa vector $\bar{b}$
- **c_exp** – **[in]** Exponent associated with input mantissa vector $\bar{c}$
- **b_hr** – **[in]** Headroom of input mantissa vector $\bar{b}$
- **c_hr** – **[in]** Headroom of input mantissa vector $\bar{c}$

void **vect_complex_s16_real_mul_prepare**(*exponent_t* *a_exp, *right_shift_t* *a_shr, const *exponent_t* b_exp, const *exponent_t* c_exp, const *headroom_t* b_hr, const *headroom_t* c_hr)

Obtain the output exponent and output shift used by *vect_complex_s16_real_mul()*.

This function is used in conjunction with *vect_complex_s16_real_mul()* to perform a complex element-wise multiplication of a complex 16-bit BFP vector by a real 16-bit vector.

This function computes `a_exp` and `a_shr`.

`a_exp` is the exponent associated with mantissa vector $\bar{a}$, and must be chosen to be large enough to avoid overflow when elements of $\bar{a}$ are computed. To maximize precision, this function chooses `a_exp` to be the smallest exponent known to avoid saturation (see exception below). The `a_exp` chosen by this function is derived from the exponents and headrooms of associated with the input vectors.

`a_shr` is the shift parameter required by *vect_complex_s16_real_mul()* to achieve the chosen output exponent `a_exp`.

`b_exp` and `c_exp` are the exponents associated with the input mantissa vectors $\bar{b}$ and $\bar{c}$ respectively.

`b_hr` and `c_hr` are the headroom of $\bar{b}$ and $\bar{c}$ respectively. If the headroom of $\bar{b}$ or $\bar{c}$ is unknown, they can be obtained by calling *vect_complex_s16_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent `desired_exp` is needed for the result (e.g. for emulating fixed-point arithmetic), the `a_shr` and `c_shr` produced by this function can be adjusted according to the following:

```
exponent_t desired_exp = ...; // Value known a priori
right_shift_t new_a_shr = a_shr + (desired_exp - a_exp);
```

When applying the above adjustment, the following conditions should be maintained:

- `new_a_shr >= 0`

Be aware that using smaller values than strictly necessary for `a_shr` can result in saturation, and using larger values may result in unnecessary underflows or loss of precision.

Notes

- Using the outputs of this function, an output mantissa which would otherwise be `INT16_MIN` will instead saturate to `-INT16_MAX`. This is due to the symmetric saturation logic employed by the VPU and is a hardware feature. This is a corner case which is usually unlikely and results in 1 LSb of error when it occurs.

See also:

vect_complex_s16_real_mul

Parameters

- **`a_exp`** – **[out]** Exponent associated with output mantissa vector $\bar{a}$
- **`a_shr`** – **[out]** Unsigned arithmetic right-shift for $\bar{a}$ used by *vect_complex_s16_real_mul()*
- **`b_exp`** – **[in]** Exponent associated with input mantissa vector $\bar{b}$
- **`c_exp`** – **[in]** Exponent associated with input mantissa vector $\bar{c}$
- **`b_hr`** – **[in]** Headroom of input mantissa vector $\bar{b}$
- **`c_hr`** – **[in]** Headroom of input mantissa vector $\bar{c}$

void **`vect_complex_s16_squared_mag_prepare`**(*exponent_t* *a_exp, *right_shift_t* *a_shr, const *exponent_t* b_exp, const *headroom_t* b_hr)

Obtain the output exponent and input shift used by *vect_complex_s16_squared_mag()*.

This function is used in conjunction with *vect_complex_s16_squared_mag()* to compute the squared magnitude of each element of a complex 16-bit BFP vector.

This function computes `a_exp` and `a_shr`.

`a_exp` is the exponent associated with mantissa vector $\bar{a}$, and is be chosen to maximize precision when elements of $\bar{a}$ are computed. The `a_exp` chosen by this function is derived from the exponent and headroom associated with the input vector.

`a_shr` is the shift parameter required by *vect_complex_s16_mag()* to achieve the chosen output exponent `a_exp`.

`b_exp` is the exponent associated with the input mantissa vector $\bar{b}$.

`b_hr` is the headroom of $\bar{b}$. If the headroom of $\bar{b}$ is unknown it can be calculated using *vect_complex_s16_headroom()*. Alternatively, the value `0` can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent `desired_exp` is needed for the result (e.g. for emulating fixed-point arithmetic), the `a_shr` produced by this function can be adjusted according to the following:

```
exponent_t a_exp;
right_shift_t a_shr;
vect_s16_mul_prepare(&a_exp, &a_shr, b_exp, c_exp, b_hr, c_hr);
exponent_t desired_exp = ...; // Value known a priori
a_shr = a_shr + (desired_exp - a_exp);
a_exp = desired_exp;
```

When applying the above adjustment, the following condition should be maintained:

- `a_shr >= 0`

Using larger values than strictly necessary for `a_shr` may result in unnecessary underflows or loss of precision.

See also:

vect_complex_s16_squared_mag()

Parameters

- **`a_exp`** – **[out]** Output exponent associated with output mantissa vector $\bar{a}$
- **`a_shr`** – **[out]** Unsigned arithmetic right-shift for $\bar{a}$ used by *vect_complex_s16_squared_mag()*
- **`b_exp`** – **[in]** Exponent associated with input mantissa vector $\bar{b}$
- **`b_hr`** – **[in]** Headroom of input mantissa vector $\bar{b}$

4.7.12 32-Bit Complex Vector Prepare Functions

group **vect_complex_s32_prepare_api**

Defines

vect_complex_s32_add_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s32_add()*.

The logic for computing the shifts and exponents of *vect_complex_s32_add()* is identical to that for *vect_s32_add()*.

This macro is provided as a convenience to developers and to make the code more coherent.

See also:

vect_s32_add_prepare()

vect_complex_s32_add_scalar_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s32_add_scalar()*.

The logic for computing the shifts and exponents of *vect_complex_s32_add_scalar()* is identical to that for *vect_s32_add()*.

This macro is provided as a convenience to developers and to make the code more readable.

See also:

vect_s32_add_prepare()

vect_complex_s32_conj_mul_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s32_conj_mul()*.

The logic for computing the shifts and exponents of *vect_complex_s32_conj_mul()* is identical to that for *vect_complex_s32_mul()*.

This macro is provided as a convenience to developers and to make the code more readable.

See also:

vect_complex_s32_mul_prepare()

vect_complex_s32_nmacc_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s32_nmacc()*.

The logic for computing the shifts and exponents of *vect_complex_s32_nmacc()* is identical to that for *vect_complex_s32_macc_prepare()*.

This macro is provided as a convenience to developers and to make the code more readable.

See also:

vect_complex_s32_macc_prepare(), *vect_complex_s32_nmacc()*

vect_complex_s32_conj_macc_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s32_conj_macc()*.

The logic for computing the shifts and exponents of *vect_complex_s32_conj_macc()* is identical to that for *vect_complex_s32_macc_prepare()*.

This macro is provided as a convenience to developers and to make the code more readable.

vect_complex_s32_macc_prepare(), *vect_complex_s32_conj_macc()*

vect_complex_s32_conj_nmacc_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s32_conj_nmacc()*.

The logic for computing the shifts and exponents of *vect_complex_s32_conj_nmacc()* is identical to that for *vect_complex_s32_macc_prepare()*.

This macro is provided as a convenience to developers and to make the code more readable.

See also:

vect_complex_s32_macc_prepare(), *vect_complex_s32_conj_nmacc()*

vect_complex_s32_real_scale_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s32_real_scale()*.

The logic for computing the shifts and exponents of *vect_complex_s32_real_scale()* is identical to that for *vect_s32_mul()*.

This macro is provided as a convenience to developers and to make the code more readable.

See also:

vect_s32_mul_prepare()

vect_complex_s32_sub_prepare

Obtain the output exponent and shifts required for a call to *vect_complex_s32_sub()*.

The logic for computing the shifts and exponents of *vect_complex_s32_sub()* is identical to that for *vect_s32_add()*.

This macro is provided as a convenience to developers and to make the code more readable.

See also:

vect_s32_add_prepare()

Functions

void **vect_complex_s32_macc_prepare**(*exponent_t* *new_acc_exp, *right_shift_t* *acc_shr, *right_shift_t* *b_shr, *right_shift_t* *c_shr, const *exponent_t* acc_exp, const *exponent_t* b_exp, const *exponent_t* c_exp, const *exponent_t* acc_hr, const *headroom_t* b_hr, const *headroom_t* c_hr)

Obtain the output exponent and shifts needed by *vect_complex_s32_macc()*.

This function is used in conjunction with *vect_complex_s32_macc()* to perform an element-wise multiply-accumlate of 32-bit BFP vectors.

This function computes new_acc_exp, acc_shr, b_shr and c_shr, which are selected to maximize precision in the resulting accumulator vector without causing saturation of final or intermediate values. Normally the caller will pass these outputs to their corresponding inputs of *vect_complex_s32_macc()*.

acc_exp is the exponent associated with the accumulator mantissa vector $\bar{a}$ prior to the operation, whereas new_acc_exp is the exponent corresponding to the updated accumulator vector.

b_exp and c_exp are the exponents associated with the complex input mantissa vectors $\bar{b}$ and $\bar{c}$ respectively.

acc_hr, b_hr and c_hr are the headrooms of $\bar{a}, \bar{b}$ and $\bar{c}$ respectively. If the headroom of any of these vectors is unknown, it can be obtained by calling *vect_complex_s32_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent desired_exp is needed for the result (e.g. for emulating fixed-point arithmetic), the acc_shr and bc_sat produced by this function can be adjusted according to the following:

```
// Presumed to be set somewhere
exponent_t acc_exp, b_exp, c_exp;
headroom_t acc_hr, b_hr, c_hr;
exponent_t desired_exp;

...

// Call prepare
right_shift_t acc_shr, b_shr, c_shr;
vect_complex_s32_macc_prepare(&acc_exp, &acc_shr, &b_shr, &c_shr,
                              acc_exp, b_exp, c_exp,
                              acc_hr, b_hr, c_hr);

// Modify results
right_shift_t mant_shr = desired_exp - acc_exp;
acc_exp += mant_shr;
acc_shr += mant_shr;
b_shr  += mant_shr;
c_shr  += mant_shr;

// acc_shr, b_shr and c_shr may now be used in a call to vect_complex_s32_macc()
```

When applying the above adjustment, the following conditions should be maintained:

- acc_shr > -acc_hr (Shifting any further left may cause saturation)
- b_shr => -b_hr (Shifting any further left may cause saturation)
- c_shr => -c_hr (Shifting any further left may cause saturation)

It is up to the user to ensure any such modification does not result in saturation or unacceptable loss of precision.

See also:

vect_complex_s32_macc

Parameters

- **new_acc_exp** − **[out]** Exponent associated with output mantissa vector $\bar{a}$ (after macc)
- **acc_shr** − **[out]** Signed arithmetic right-shift used for $\bar{a}$ in *vect_complex_s32_macc()*
- **b_shr** − **[out]** Signed arithmetic right-shift used for $\bar{b}$ in *vect_complex_s32_macc()*
- **c_shr** − **[out]** Signed arithmetic right-shift used for $\bar{c}$ in *vect_complex_s32_macc()*

- **acc_exp** − **[in]** Exponent associated with input mantissa vector $\bar{a}$ (before macc)

- **b_exp** − **[in]** Exponent associated with input mantissa vector $\bar{b}$

- **c_exp** − **[in]** Exponent associated with input mantissa vector $\bar{c}$

- **acc_hr** − **[in]** Headroom of input mantissa vector $\bar{a}$ (before macc)

- **b_hr** − **[in]** Headroom of input mantissa vector $\bar{b}$

- **c_hr** − **[in]** Headroom of input mantissa vector $\bar{c}$

void **vect_complex_s32_mag_prepare**(*exponent_t* *a_exp, *right_shift_t* *b_shr, const *exponent_t* b_exp, const *headroom_t* b_hr)

Obtain the output exponent and input shift used by *vect_complex_s32_mag()* and *vect_complex_s16_mag()*.

This function is used in conjunction with *vect_complex_s32_mag()* to compute the magnitude of each element of a complex 32-bit BFP vector.

This function computes a_exp and b_shr.

a_exp is the exponent associated with mantissa vector $\bar{a}$, and is be chosen to maximize precision when elements of $\bar{a}$ are computed. The a_exp chosen by this function is derived from the exponent and headroom associated with the input vector.

b_shr is the shift parameter required by *vect_complex_s32_mag()* to achieve the chosen output exponent a_exp.

b_exp is the exponent associated with the input mantissa vector $\bar{b}$.

b_hr is the headroom of $\bar{b}$. If the headroom of $\bar{b}$ is unknown it can be calculated using *vect_complex_s32_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent desired_exp is needed for the result (e.g. for emulating fixed-point arithmetic), the b_shr produced by this function can be adjusted according to the following:

```
exponent_t desired_exp = ...; // Value known a priori
right_shift_t new_b_shr = b_shr + (desired_exp - a_exp);
```

When applying the above adjustment, the following condition should be maintained:

- b_hr + b_shr >= 0

Using larger values than strictly necessary for b_shr may result in unnecessary underflows or loss of precision.

See also:

vect_complex_s32_mag()

Parameters

- **a_exp** − **[out]** Output exponent associated with output mantissa vector $\bar{a}$

- **b_shr** − **[out]** Signed arithmetic right-shift for $\bar{b}$ used by *vect_complex_s32_mag()*

- **b_exp** − **[in]** Exponent associated with input mantissa vector $\bar{b}$

- **b_hr** − **[in]** Headroom of input mantissa vector $\bar{b}$

void **vect_complex_s32_mul_prepare**(*exponent_t* *a_exp, *right_shift_t* *b_shr, *right_shift_t* *c_shr, const *exponent_t* b_exp, const *exponent_t* c_exp, const *headroom_t* b_hr, const *headroom_t* c_hr)

Obtain the output exponent and input shifts used by *vect_complex_s32_mul()* and *vect_complex_s32_conj_mul()*.

This function is used in conjunction with *vect_complex_s32_mul()* to perform a complex element-wise multiplication of two complex 32-bit BFP vectors.

This function computes `a_exp`, `b_shr` and `c_shr`.

`a_exp` is the exponent associated with mantissa vector $\bar{a}$, and must be chosen to be large enough to avoid overflow when elements of $\bar{a}$ are computed. To maximize precision, this function chooses `a_exp` to be the smallest exponent known to avoid saturation (see exception below). The `a_exp` chosen by this function is derived from the exponents and headrooms of associated with the input vectors.

`b_shr` and `c_shr` are the shift parameters required by *vect_complex_s32_mul()* to achieve the chosen output exponent `a_exp`.

`b_exp` and `c_exp` are the exponents associated with the input mantissa vectors $\bar{b}$ and $\bar{c}$ respectively.

`b_hr` and `c_hr` are the headroom of $\bar{b}$ and $\bar{c}$ respectively. If the headroom of $\bar{b}$ or $\bar{c}$ is unknown, they can be obtained by calling *vect_complex_s32_headroom()*. Alternatively, the value `0` can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent `desired_exp` is needed for the result (e.g. for emulating fixed-point arithmetic), the `b_shr` and `c_shr` produced by this function can be adjusted according to the following:

```
exponent_t desired_exp = ...; // Value known a priori
right_shift_t new_b_shr = b_shr + (desired_exp - a_exp);
right_shift_t new_c_shr = c_shr + (desired_exp - a_exp);
```

When applying the above adjustment, the following conditions should be maintained:

- `b_hr + b_shr >= 0`
- `c_hr + c_shr >= 0`

Be aware that using smaller values than strictly necessary for `b_shr` and `c_shr` can result in saturation, and using larger values may result in unnecessary underflows or loss of precision.

Notes

- Using the outputs of this function, an output mantissa which would otherwise be `INT32_MIN` will instead saturate to `-INT32_MAX`. This is due to the symmetric saturation logic employed by the VPU and is a hardware feature. This is a corner case which is usually unlikely and results in 1 LSb of error when it occurs.

See also:

vect_complex_s32_conj_mul, *vect_complex_s32_mul*

Parameters

- **a_exp** – **[out]** Exponent associated with output mantissa vector $\bar{a}$

- **b_shr** – **[out]** Signed arithmetic right-shift for $\bar{b}$ used by *vect_complex_s32_mul()*

- **c_shr** – **[out]** Signed arithmetic right-shift for $\bar{c}$ used by *vect_complex_s32_mul()*

- **b_exp** – **[in]** Exponent associated with input mantissa vector $\bar{b}$

- **c_exp** – **[in]** Exponent associated with input mantissa vector $\bar{c}$

- **b_hr** – **[in]** Headroom of input mantissa vector $\bar{b}$

- **c_hr** – **[in]** Headroom of input mantissa vector $\bar{c}$

void **vect_complex_s32_real_mul_prepare**(*exponent_t* *a_exp, *right_shift_t* *b_shr, *right_shift_t* *c_shr, const *exponent_t* b_exp, const *exponent_t* c_exp, const *headroom_t* b_hr, const *headroom_t* c_hr)

Obtain the output exponent and input shifts used by *vect_complex_s32_real_mul()*.

This function is used in conjunction with *vect_complex_s32_real_mul()* to perform a the element-wise multiplication of complex 32-bit BFP vector by a real 32-bit BFP vector.

This function computes a_exp, b_shr and c_shr.

a_exp is the exponent associated with mantissa vector $\bar{a}$, and must be chosen to be large enough to avoid overflow when elements of $\bar{a}$ are computed. To maximize precision, this function chooses a_exp to be the smallest exponent known to avoid saturation (see exception below). The a_exp chosen by this function is derived from the exponents and headrooms of associated with the input vectors.

b_shr and c_shr are the shift parameters required by *vect_complex_s32_mul()* to achieve the chosen output exponent a_exp.

b_exp and c_exp are the exponents associated with the input mantissa vectors $\bar{b}$ and $\bar{c}$ respectively.

b_hr and c_hr are the headroom of $\bar{b}$ and $\bar{c}$ respectively. If the headroom of $\bar{b}$ or $\bar{c}$ is unknown, they can be obtained by calling *vect_complex_s32_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent desired_exp is needed for the result (e.g. for emulating fixed-point arithmetic), the b_shr and c_shr produced by this function can be adjusted according to the following:

```
exponent_t desired_exp = ...; // Value known a priori
right_shift_t new_b_shr = b_shr + (desired_exp - a_exp);
right_shift_t new_c_shr = c_shr + (desired_exp - a_exp);
```

When applying the above adjustment, the following conditions should be maintained:

- b_hr + b_shr >= 0

- c_hr + c_shr >= 0

Be aware that using smaller values than strictly necessary for b_shr and c_shr can result in saturation, and using larger values may result in unnecessary underflows or loss of precision.

Notes

- Using the outputs of this function, an output mantissa which would otherwise be `INT32_MIN` will instead saturate to `-INT32_MAX`. This is due to the symmetric saturation logic employed by the VPU and is a hardware feature. This is a corner case which is usually unlikely and results in 1 LSb of error when it occurs.

See also:

vect_complex_s32_real_mul

Parameters

- **a_exp** – **[out]** Output exponent associated with $\bar{a}$
- **b_shr** – **[out]** Signed arithmetic right-shift for $\bar{b}$ used by *vect_complex_s32_real_mul()*
- **c_shr** – **[out]** Signed arithmetic right-shift for $\bar{c}$ used by *vect_complex_s32_real_mul()*
- **b_exp** – **[in]** Exponent associated with $\bar{b}$
- **c_exp** – **[in]** Exponent associated with $\bar{c}$
- **b_hr** – **[in]** Headroom of mantissa vector $\bar{b}$
- **c_hr** – **[in]** Headroom of mantissa vector $\bar{c}$

void **vect_complex_s32_scale_prepare**(*exponent_t* *a_exp, *right_shift_t* *b_shr, *right_shift_t* *c_shr, const *exponent_t* b_exp, const *exponent_t* c_exp, const *headroom_t* b_hr, const *headroom_t* c_hr)

Obtain the output exponent and input shifts used by *vect_complex_s32_scale()*.

This function is used in conjunction with *vect_complex_s32_scale()* to perform a complex multiplication of a complex 32-bit BFP vector by a complex 32-bit scalar.

This function computes `a_exp`, `b_shr` and `c_shr`.

`a_exp` is the exponent associated with mantissa vector $\bar{a}$, and must be chosen to be large enough to avoid overflow when elements of $\bar{a}$ are computed. To maximize precision, this function chooses `a_exp` to be the smallest exponent known to avoid saturation (see exception below). The `a_exp` chosen by this function is derived from the exponents and headrooms associated with the input vectors.

`b_shr` and `c_shr` are the shift parameters required by *vect_complex_s32_mul()* to achieve the chosen output exponent `a_exp`.

`b_exp` and `c_exp` are the exponents associated with the input mantissa vectors $\bar{b}$ and $\bar{c}$ respectively.

`b_hr` and `c_hr` are the headroom of $\bar{b}$ and $\bar{c}$ respectively. If the headroom of $\bar{b}$ or $\bar{c}$ is unknown, they can be obtained by calling *vect_complex_s32_headroom()*. Alternatively, the value `0` can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent `desired_exp` is needed for the result (e.g. for emulating fixed-point arithmetic), the `b_shr` and `c_shr` produced by this function can be adjusted according to the following:

```
exponent_t desired_exp = ...; // Value known a priori
right_shift_t new_b_shr = b_shr + (desired_exp - a_exp);
right_shift_t new_c_shr = c_shr + (desired_exp - a_exp);
```

When applying the above adjustment, the following conditions should be maintained:

- `b_hr + b_shr >= 0`
- `c_hr + c_shr >= 0`

Be aware that using smaller values than strictly necessary for `b_shr` and `c_shr` can result in saturation, and using larger values may result in unnecessary underflows or loss of precision.

Notes

- Using the outputs of this function, an output mantissa which would otherwise be `INT32_MIN` will instead saturate to `-INT32_MAX`. This is due to the symmetric saturation logic employed by the VPU and is a hardware feature. This is a corner case which is usually unlikely and results in 1 LSb of error when it occurs.

See also:

vect_complex_s32_scale

Parameters

- **a_exp** − **[out]** Exponent associated with output mantissa vector $\bar{a}$
- **b_shr** − **[out]** Signed arithmetic right-shift for $\bar{b}$ used by *vect_complex_s32_scale()*
- **c_shr** − **[out]** Signed arithmetic right-shift for $\bar{c}$ used by *vect_complex_s32_scale()*
- **b_exp** − **[in]** Exponent associated with input mantissa vector $\bar{b}$
- **c_exp** − **[in]** Exponent associated with input mantissa vector $\bar{c}$
- **b_hr** − **[in]** Headroom of input mantissa vector $\bar{b}$
- **c_hr** − **[in]** Headroom of input mantissa vector $\bar{c}$

void **vect_complex_s32_squared_mag_prepare**(*exponent_t* *a_exp, *right_shift_t* *b_shr, const *exponent_t* b_exp, const *headroom_t* b_hr)

Obtain the output exponent and input shift used by *vect_complex_s32_squared_mag()*.

This function is used in conjunction with *vect_complex_s32_squared_mag()* to compute the squared magnitude of each element of a complex 32-bit BFP vector.

This function computes `a_exp` and `b_shr`.

`a_exp` is the exponent associated with mantissa vector $\bar{a}$, and is be chosen to maximize precision when elements of $\bar{a}$ are computed. The `a_exp` chosen by this function is derived from the exponent and headroom associated with the input vector.

`b_shr` is the shift parameter required by *vect_complex_s32_mag()* to achieve the chosen output exponent `a_exp`.

`b_exp` is the exponent associated with the input mantissa vector $\bar{b}$.

`b_hr` is the headroom of $\bar{b}$. If the headroom of $\bar{b}$ is unknown it can be calculated using *vect_complex_s32_headroom()*. Alternatively, the value `0` can always be safely used (but may result in reduced precision).

Adjusting Output Exponents

If a specific output exponent `desired_exp` is needed for the result (e.g. for emulating fixed-point arithmetic), the `b_shr` produced by this function can be adjusted according to the following:

```
exponent_t desired_exp = ...; // Value known a priori
right_shift_t new_b_shr = b_shr + (desired_exp - a_exp);
```

When applying the above adjustment, the following condition should be maintained:

• `b_hr + b_shr >= 0`

Using larger values than strictly necessary for `b_shr` may result in unnecessary underflows or loss of precision.

See also:

vect_complex_s32_squared_mag()

Parameters

- **`a_exp`** − **[out]** Output exponent associated with output mantissa vector $\bar{a}$
- **`b_shr`** − **[out]** Signed arithmetic right-shift for $\bar{b}$ used by *vect_complex_s32_squared_mag()*
- **`b_exp`** − **[in]** Exponent associated with input mantissa vector $\bar{b}$
- **`b_hr`** − **[in]** Headroom of input mantissa vector $\bar{b}$

void **vect_complex_s32_sum_prepare**(*exponent_t* *a_exp, *right_shift_t* *b_shr, const *exponent_t* b_exp, const *headroom_t* b_hr, const unsigned length)

Obtain the output exponent and input shift used by *vect_complex_s32_sum()*.

This function is used in conjunction with *vect_complex_s32_sum()* to compute the sum of elements of a complex 32-bit BFP vector.

This function computes `a_exp` and `b_shr`.

`a_exp` is the exponent associated with the 64-bit mantissa a returned by *vect_complex_s32_sum()*, and must be chosen to be large enough to avoid saturation when a is computed. To maximize precision, this function chooses `a_exp` to be the smallest exponent known to avoid saturation (see exception below). The `a_exp` chosen by this function is derived from the exponents and headrooms associated with the input vector.

`b_shr` is the shift parameter required by *vect_complex_s32_sum()* to achieve the chosen output exponent `a_exp`.

`b_exp` is the exponent associated with the input mantissa vector $\bar{b}$.

`b_hr` is the headroom of $\bar{b}$. If the headroom of $\bar{b}$ is unknown it can be calculated using *vect_complex_s32_headroom()*. Alternatively, the value 0 can always be safely used (but may result in reduced precision).

`length` is the number of elements in the input mantissa vector $\bar{b}$.

Adjusting Output Exponents

If a specific output exponent `desired_exp` is needed for the result (e.g. for emulating fixed-point arithmetic), the `b_shr` produced by this function can be adjusted according to the following:

```
exponent_t desired_exp = ...; // Value known a priori
right_shift_t new_b_shr = b_shr + (desired_exp - a_exp);
```

When applying the above adjustment, the following conditions should be maintained:

- `b_hr + b_shr >= 0`

Be aware that using smaller values than strictly necessary for `b_shr` can result in saturation, and using larger values may result in unnecessary underflows or loss of precision.

See also:

vect_complex_s32_sum

Parameters

- **`a_exp`** – **[out]** Exponent associated with output mantissa a
- **`b_shr`** – **[out]** Signed arithmetic right-shift for $\bar{b}$ used by *vect_complex_s32_sum()*
- **`b_exp`** – **[in]** Exponent associated with input mantissa vector $\bar{b}$
- **`b_hr`** – **[in]** Headroom of input mantissa vector $\bar{b}$
- **`length`** – **[in]** Number of elements in $\bar{b}$

4.7.13 32-Bit Vector Chunk (8-Element) API

group **chunk32_api**

Functions

int32_t **chunk_s32_dot**(const int32_t b[VPU_INT32_EPV], const *q2_30* c[VPU_INT32_EPV])

Compute the inner product between two vector chunks.

This function computes the inner product of two vector chunks, $\bar{b}$ and $\bar{c}$.

Conceptually, elements of $\bar{b}$ may have any number of fractional bits (int, fixed-point, mantissas of a BFP vector) so long as they're all the same. Elements of $\bar{c}$ are Q2.30 fixed-point values. Given that, the returned value a will have the same number of fractional bits as $\bar{b}$.

Only the lowest 32 bits of the sum a are returned.

Operation Performed

$$a \leftarrow \sum_{k=0}^{VPU\_INT32\_EPV-1} \left(round \left(\frac{b_k \cdot c_k}{2^{30}} \right) \right)$$

Parameters

- **`b`** – **[in]** Input chunk $\bar{b}$
- **`c`** – **[in]** Input chunk $\bar{c}$

Returns

a

void **chunk_s32_log**(*q8_24* a[VPU_INT32_EPV], const int32_t b[VPU_INT32_EPV], const *exponent_t* b_exp)

Compute the natural log of a vector chunk of 32-bit values.

This function computes the natural logarithm of each of the 8 elements in vector chunk $\bar{b}$. The result is returned as an 8-element chunk $\bar{a}$ of Q8.24 values.

b_exp is the exponent associated with elements of $\bar{b}$.

Any input $b_k \leq 0$ will result in a corresponding output $a_k = \text{INT32}_\text{M}\text{IN}$.

Operation Performed

$$a_k \leftarrow \begin{cases} log(b_k \cdot 2^{\text{b\_exp}}) & b_k > 0 \\ \text{INT32\_MIN} & \text{otherwise} \end{cases}$$
$$\text{for } k \in 0..\text{VPU\_INT32\_EPV} - 1$$

Parameters

- **a** − **[out]** Output vector chunk $\bar{a}$
- **b** − **[in]** Input vector chunk $\bar{b}$
- **b_exp** − **[in]** Exponent associated with $\bar{b}$

Throws ET_LOAD_STORE
Raised if b or a is not double word-aligned (See *Note: Vector Alignment*)

void **chunk_float_s32_log**(*q8_24* a[VPU_INT32_EPV], const *float_s32_t* b[VPU_INT32_EPV])

Compute the natural log of a vector chunk of `float_s32_t`.

This function computes the natural logarithm of each of the VPU_INT32_EPV elements in vector chunk $\bar{b}$. The result is returned as an 8-element chunk $\bar{a}$ of Q8.24 values.

Any input $b_k \leq 0$ will result in a corresponding output $a_k = \text{INT32}_\text{M}\text{IN}$.

Operation Performed

$$a_k \leftarrow \begin{cases} log(b_k) & b_k > 0 \\ \text{INT32\_MIN} & \text{otherwise} \end{cases}$$
$$\text{for } k \in 0..\text{VPU\_INT32\_EPV} - 1$$

Parameters

- **a** − **[out]** Output vector chunk $\bar{a}$
- **b** − **[in]** Input vector chunk $\bar{b}$

Throws ET_LOAD_STORE
Raised if b or a is not double word-aligned (See *Note: Vector Alignment*)

void **chunk_q30_power_series**(int32_t a[VPU_INT32_EPV], const *q2_30* b[VPU_INT32_EPV], const int32_t c[], const unsigned term_count)

Compute a power series on a vector chunk of Q2.30 values.

This function is used to compute a power series summation on a vector chunk (VPU_INT32_EPV-element vector) $\bar{b}$. $\bar{b}$ contains Q2.30 values. $\bar{c}$ is a vector containing coefficients to be multiplied

by powers of $\bar{b}$, and may have any associated exponent. The output is vector chunk $\bar{a}$ and has the same exponent as $\bar{c}$.

`c[]` is an array with shape (`term_count`, `VPU_INT32_EPV`), where the second axis contains the same value replicated across all `VPU_INT32_EPV` elements. That is, `c[k][i] = c[k][j]` for `i` and `j` in `0..(VPU_INT32_EPV-1)`. This is for performance reasons. (For the purpose of this explanation, $\bar{c}$ is considered to be single-dimensional, without redundancy.)

Operation Performed

$$b_{k,0} = 2^{30}$$

$$b_{k,i} = round\left(\frac{b_{k,i-1} \cdot b_k}{2^{30}}\right)$$

$$\text{for } i \in 1..(N-1)$$

$$a_k \leftarrow \sum_{i=0}^{N-1} round\left(\frac{b_{k,i} \cdot c_i}{2^{30}}\right)$$

$$\text{for } k \in 0..\text{VPU\_INT32\_EPV} - 1$$

Parameters

- **a** – **[out]** Output vector chunk $\bar{a}$
- **b** – **[in]** Input vector chunk $\bar{b}$
- **c** – **[in]** Coefficient vector $\bar{c}$
- **term_count** – **[in]** Number of power series terms, N

void **chunk_q30_exp_small**(*q2_30* a[VPU_INT32_EPV], const *q2_30* b[VPU_INT32_EPV])

Compute e^b on a vector chunk of Q2.30 values.

This function computes e^{b_k} for each element of a vector chunk (`VPU_INT32_EPV`-element vector) $\bar{b}$ of Q2.30 values near 0. The result is computed using the power series approximation of e^x near zero. It is recommended that this function only be used for $-0.5 \leq b_k \cdot 2^{-30} \leq 0.5$.

The output vector chunk $\bar{a}$ is also in a Q2.30 format.

Operation Performed

$$a_k \leftarrow e^{b_k \cdot 2^{-30}}$$

$$\text{for } k \in 0..\text{VPU\_INT32\_EPV}$$

Parameters

- **a** – **[out]** Output vector chunk $\bar{a}$
- **b** – **[in]** Input vector chunk $\bar{b}$

4.8 Q-Format Macros

group `qfmt_macros`

Defines

F(N)

    Convert fixed-point value to double-precision float.

    This macro is meant to allow for parameterized access to the more specific conversion macros, such as *F8()*, *F24()*, *F31()* and so on. Being parameterized allows the user to specify the Q-format (fractional bit count) using another macro. For example:

```
#define X_FRAC_BITS   24
int32_t x = ...;
...
// Convert x to double
double dbl_x = F(X_FRAC_BITS)(x);
```

Q(N)

    Convert floating-point value to fixed-point value.

    This macro is meant to allow for parameterized access to the more specific conversion macros, such as *Q8()*, *Q24()*, *Q31()* and so on. Being parameterized allows the user to specify the Q-format (fractional bit count) using another macro. For example:

```
#include <math.h>
...
#define PI_FRAC_BITS   24
int32_t x = Q(PI_FRAC_BITS)(M_PI);
```

Q31(f)

    Convert `double` value to Q1.31 fixed-point value, with rounding.

Q30(f)

    Convert `double` value to Q2.30 fixed-point value, with rounding.

Q29(f)

    Convert `double` value to Q3.29 fixed-point value, with rounding.

Q28(f)

    Convert `double` value to Q4.28 fixed-point value, with rounding.

Q27(f)

    Convert `double` value to Q5.27 fixed-point value, with rounding.

Q26(f)

    Convert `double` value to Q6.26 fixed-point value, with rounding.

Q25(f)

    Convert `double` value to Q7.25 fixed-point value, with rounding.

Q24(f)

    Convert `double` value to Q8.24 fixed-point value, with rounding.

Q23(f)

 Convert `double` value to Q9.23 fixed-point value, with rounding.

Q22(f)

 Convert `double` value to Q10.22 fixed-point value, with rounding.

Q21(f)

 Convert `double` value to Q11.21 fixed-point value, with rounding.

Q20(f)

 Convert `double` value to Q12.20 fixed-point value, with rounding.

Q19(f)

 Convert `double` value to Q13.19 fixed-point value, with rounding.

Q18(f)

 Convert `double` value to Q14.18 fixed-point value, with rounding.

Q17(f)

 Convert `double` value to Q15.17 fixed-point value, with rounding.

Q16(f)

 Convert `double` value to Q16.16 fixed-point value, with rounding.

Q15(f)

 Convert `double` value to Q17.15 fixed-point value, with rounding.

Q14(f)

 Convert `double` value to Q18.14 fixed-point value, with rounding.

Q13(f)

 Convert `double` value to Q19.13 fixed-point value, with rounding.

Q12(f)

 Convert `double` value to Q20.12 fixed-point value, with rounding.

Q11(f)

 Convert `double` value to Q21.11 fixed-point value, with rounding.

Q10(f)

 Convert `double` value to Q22.10 fixed-point value, with rounding.

Q9(f)

 Convert `double` value to Q23.9 fixed-point value, with rounding.

Q8(f)

 Convert `double` value to Q24.8 fixed-point value, with rounding.

F31(x)

 Convert Q1.31 fixed-point value to `double` value.

F30(x)

 Convert Q2.30 fixed-point value to `double` value.

F29(x)

 Convert Q3.29 fixed-point value to `double` value.

F28(x)

 Convert Q4.28 fixed-point value to `double` value.

F27(x)

 Convert Q5.27 fixed-point value to `double` value.

F26(x)

Convert Q6.26 fixed-point value to `double` value.

F25(x)

Convert Q7.25 fixed-point value to `double` value.

F24(x)

Convert Q8.24 fixed-point value to `double` value.

F23(x)

Convert Q9.23 fixed-point value to `double` value.

F22(x)

Convert Q10.22 fixed-point value to `double` value.

F21(x)

Convert Q11.21 fixed-point value to `double` value.

F20(x)

Convert Q12.20 fixed-point value to `double` value.

F19(x)

Convert Q13.19 fixed-point value to `double` value.

F18(x)

Convert Q14.18 fixed-point value to `double` value.

F17(x)

Convert Q15.17 fixed-point value to `double` value.

F16(x)

Convert Q16.16 fixed-point value to `double` value.

F15(x)

Convert Q17.15 fixed-point value to `double` value.

F14(x)

Convert Q18.14 fixed-point value to `double` value.

F13(x)

Convert Q19.13 fixed-point value to `double` value.

F12(x)

Convert Q20.12 fixed-point value to `double` value.

F11(x)

Convert Q21.11 fixed-point value to `double` value.

F10(x)

Convert Q22.10 fixed-point value to `double` value.

F9(x)

Convert Q23.9 fixed-point value to `double` value.

F8(x)

Convert Q24.8 fixed-point value to `double` value.

4.9 XMath Util Functions and Macros

group `util_macros`

Defines

MAX(A, B)

    Takes the greater of arguments A and B, preferring A on equality.

> **Note:** This is not safe from multiple evaluation of arguments.

    Parameters

- **A** – **[in]** First input
- **B** – **[in]** Second input

    Returns
        Maximum of the inputs.

MIN(A, B)

    Takes the lesser of arguments A and B, preferring A on equality.

> **Note:** This is not safe from multiple evaluation of arguments.

    Parameters

- **A** – **[in]** First input
- **B** – **[in]** Second input

    Returns
        Minimum of the inputs.

CLS_S8(X)

    Count leading sign bits of an `int8_t`.

    Parameters

- **X** – **[in]** Input

    Returns
        Leading sign bits of X

CLS_S16(X)

    Count leading sign bits of an `int16_t`.

    Parameters

- **X** – **[in]** Input

    Returns
        Leading sign bits of X

CLS_S32(X)

    Count leading sign bits of an `int32_t`.

    Parameters

- **x** – **[in]** Input

> **Returns**
> Leading sign bits of X

CLS_S64(X)

Count leading sign bits of an `int64_t`.

> **Parameters**
>
> - **x** – **[in]** Input
>
> **Returns**
> Leading sign bits of X

CLS_C16(X)

Count leading sign bits of a *`complex_s16_t`*.

The number of leading sign bits for a complex integer is defined as the minimum of the number of leading sign bits for its real part and for its imaginary part.

> **Parameters**
>
> - **x** – **[in]** Input
>
> **Returns**
> Leading sign bits of X

CLS_C32(X)

Count leading sign bits of a *`complex_s32_t`*.

The number of leading sign bits for a complex integer is defined as the minimum of the number of leading sign bits for its real part and for its imaginary part.

> **Parameters**
>
> - **x** – **[in]** Input
>
> **Returns**
> Leading sign bits of X

HR_S64(X)

Get the headroom of an `int64_t`.

> **Parameters**
>
> - **x** – **[in]** Input
>
> **Returns**
> Headroom of X

HR_S32(X)

Get the headroom of an `int32_t`.

> **Parameters**
>
> - **x** – **[in]** Input
>
> **Returns**
> Headroom of X

HR_S16(X)

Get the headroom of an `int16_t`.

> **Parameters**
>
> - **x** – **[in]** Input
>
> **Returns**
> Headroom of X

HR_S8(X)

> Get the headroom of an `int8_t`.

> **Parameters**
>> • **X** – **[in]** Input

> **Returns**
>> Headroom of `X`

HR_C32(X)

> Get the headroom of a *`complex_s32_t`*.

> The headroom of a complex `N`-bit integer is the minimum of the headroom of each of its `N`-bit real and imaginary parts.

> **Parameters**
>> • **X** – **[in]** Input

> **Returns**
>> Headroom of `X`

HR_C16(X)

> Get the headroom of a *`complex_s16_t`*.

> The headroom of a complex `N`-bit integer is the minimum of the headroom of each of its `N`-bit real and imaginary parts.

> **Parameters**
>> • **X** – **[in]** Input

> **Returns**
>> Headroom of `X`

Functions

void **xs3_memcpy**(void *dst, const void *src, unsigned bytes)

> VPU-based memcpy implementation.

> Same as standard `memcpy()` except for an extra constraint that both `dst` and`src`' must be word-aligned addresses.

> **Parameters**
>> • **dst** – **[out]** Destination address
>> • **src** – **[in]** Source address
>> • **bytes** – **[in]** Number of bytes to copy

static inline unsigned **cls**(const int32_t a)

> Count leading sign bits of `int32_t`.

> This function returns the number of most-significant bits in `a` which are equal to its sign bit.

> **Note:** This is the total number of leading sign bits, not *redundant* leading sign bits.

> **Parameters**
>> • **a** – **[in]** Input value

> **Returns**
>> Number of leading sign bits of `a`

static inline unsigned **n_bitrev**(const unsigned index, const unsigned bits)

>Reverse the bits of an integer.

>This function returns takes an integer `index` and reverses the `bits` least-significant bits to form a new integer which is returned. All more significant bits are ignored.

>This is useful for algorithms, such as the FFT, whose implementation requires reordering of elements by reversing the bits of the indices.

>>**Parameters**
>>- `index` – **[in]** Input value
>>- `bits` – **[in]** The number of least-significant bits to reverse.

>>**Returns**
>>The `bits` LSb's of `index` reversed.

4.10 Library Configuration

4.10.1 Configuration Options

group **config_options**

Defines

XMATH_BFP_DEBUG_CHECK_LENGTHS

>Indicates whether the BFP functions should check vector lengths for errors.

>Iff true, BFP functions will check (`assert()`) to ensure that each BFP vector argument does not violate any length constraints. Most often this simply ensures that, where BFP functions take multiple vectors as parameters, each of the vectors has the same length.

>Defaults to false (`0`).

XMATH_BFP_SQRT_DEPTH_S16

>The number of most significant bits which are computed by *bfp_s16_sqrt()*.

>The function bfp_sqrt_s16() computes results one bit at a time, starting with bit 14 (the second-to-most significant bit). Because this is a relatively expensive operation, it may be desirable to trade off precision of results for a speed-up.

>The time cost of bfp_sqrt_s16() is approximately linear with respect to the depth.

>Defaults to `VECT_SQRT_S16_MAX_DEPTH` (15)

>>**See also:**
>>*bfp_s16_sqrt*

XMATH_BFP_SQRT_DEPTH_S32

>The number of most significant bits which are computed by *bfp_s32_sqrt()*.

>The function bfp_sqrt_s32() computes results one bit at a time, starting with bit 30 (the second-to-most significant bit). Because this is a relatively expensive operation, it may be desirable to trade off precision of results for a speed-up.

The time cost of bfp_sqrt_s32() is approximately linear with respect to the depth.

Defaults to `VECT_SQRT_S32_MAX_DEPTH` (31)

See also:

bfp_s32_sqrt

XMATH_MALLOC

Function used to dynamically allocate memory.

This function is used to dynamically allocate memory. Defaults to `malloc`. Must have same signature as `malloc()`

See also:

XMATH_FREE

XMATH_FREE

Function use to free dynamically allocated memory.

This function is used to deallocate dynamically allocated memory. Defaults to `free`. Must have same signature as `free()`

See also:

XMATH_MALLOC

4.11 Library Notes

4.11.1 Note: Vector Alignment

page `note_vector_alignment`

This library makes use of the XMOS architecture's vector processing unit (VPU). All loads and stores to and from the XS3 VPU have the requirement that the loaded/stored addresses must be aligned to a 4-byte boundary (word-aligned).

In the current version of the API, this leads to the requirement that most API functions require vectors (or the data backing a BFP vector) to begin at word-aligned addresses. Vectors are *not* required, however, to have a size (in bytes) that is a multiple of 4.

Writing Alignment-safe Code

The alignment requirement is ultimately always on the data that backs a vector. For the low-level API, that is the pointers passed to the functions themselves. For the high-level API, that is the memory to which the `data` field (or the `real` and `imag` fields in the case of *bfp_complex_s16_t*) points, specified when the BFP vector is initialized.

Arrays of type `int32_t` and *complex_s32_t* will normally be guaranteed to be word-aligned by the compiler. However, if the user manually specifies the beginning of an `int32_t` array, as in the following..

```
uint8_t byte_buffer[100];
int32_t* integer_array = (int32_t*) &byte_buffer[1];
```

.. the vector may not be word-aligned. It is the responsibility of the user to ensure proper alignment of data.

For int16_t arrays, the compiler does not by default guarantee that the array starts on a word-aligned address. To force word-alignment on arrays of this type, use __attribute__((aligned (4))) in the variable definition, as in the following.

```
int16_t __attribute__((aligned (4))) data[100];
```

Occasionally, 8-byte (double word) alignment is required. In this case, neither int32_t nor int16_t is necessarily guaranteed to align as required. Similar to the above, this can be hinted to the compiler as in the following.

```
int32_t __attribute__((aligned (8))) data[100];
```

This library also provides the macros WORD_ALIGNED and DWORD_ALIGNED which force 4- and 8-byte alignment respectively as above.

4.11.2   Note: Symmetrically Saturating Arithmetic

page `note_symmetric_saturation`

With ordinary integer arithmetic the block floating-point logic chooses exponents and operand shifts to prevent integer overflow with worst-case input values. However, the XS3 VPU uses symmetrically saturating integer arithmetic.

Saturating arithmetic is that where partial results of the applied operation use a bit depth greater than the output bit depth, and values that can't be properly expressed with the output bit depth are set to the nearest expressible value.

For example, in ordinary C integer arithmetic, a function which multiplies two 32-bit integers may internally compute the full 64-bit product and then clamp values to the range (INT32_MIN, INT32_MAX) before returning a 32-bit result.

Symmetrically saturating arithmetic also includes the property that the lower bound of the expressible range is the negative of the upper bound of the expressible range.

One of the major troubles with non-saturating integer arithmetic is that in a twos complement encoding, there exists a non-zero integer (e.g. INT16_MIN in 16-bit twos complement arithmetic) value x for which $-1 \cdot x = x$. Serious arithmetic errors can result when this case is not accounted for.

One of the results of *symmetric* saturation, on the other hand, is that there is a corner case where (using the same exponent and shift logic as non-saturating arithmetic) saturation may occur for a particular combination of input mantissas. The corner case is different for different operations.

When the corner case occurs, the minimum (and largest magnitude) value of the resulting vector is 1 LSb greater than its ideal value (e.g. -0x3FFF instead of -0x4000 for 16-bit arithmetic). The error in this output element's mantissa is then 1 LSb, or 2^p, where p is the exponent of the resulting BFP vector.

Of course, the very nature of BFP arithmetic routinely involves errors of this magnitude.

4.11.3 Note: Spectrum Packing

In its general form, the N-point Discrete Fourier Transform is an operation applied to a complex N-point signal $x[n]$ to produce a complex spectrum $X[f]$. Any spectrum $X[f]$ which is the result of a N-point DFT has the property that $X[f+N] = X[f]$. Thus, the complete representation of the N-point DFT of $X[n]$ requires N complex elements.

Complex DFT and IDFT

In this library, when performing a complex DFT (e.g. using fft_bfp_forward_complex()), the spectral representation that results in a straight-forward mapping:

X[f] $\longleftarrow X[f]$ for $0 \leq f < N$

where X is an N-element array of `complex_s32_t`, where the real part of $X[f]$ is in X[f].re and the imaginary part in X[f].im.

Likewise, when performing an N-point complex inverse DFT, that is also the representation that is expected.

Real DFT and IDFT

Oftentimes we instead wish to compute the DFT of real signals. In addition to the periodicity property ($X[f+N] = X[f]$), the DFT of a real signal also has a complex conjugate symmetry such that $X[-f] = X^*[f]$, where $X^*[f]$ is the complex conjugate of $X[f]$. This symmetry makes it redundant (and thus undesirable) tostore such symmetric pairs of elements. This would allow us to get away with only explicitly storing $X[f$ for $0 \leq f \leq N/2$ in $(N/2) + 1$ complex elements.

Unfortunately, using such a representation has the undesirable property that the DFT of an N-point real signal cannot be computed in-place, as the representation requires more memory than we started with.

However, if we take the periodicity and complex conjugate symmetry properties together:

$$X[0] = X^*[0] \rightarrow Imag\{X[0]\} = 0$$
$$X[-(N/2) + N] = X[N/2]$$
$$X[-N/2] = X^*[N/2] \rightarrow X[N/2] = X^*[N/2] \rightarrow Imag\{X[N/2]\} = 0$$

Because both $X[0]$ and $X[N/2]$ are guaranteed to be real, we can recover the benefit of in-place computation in our representation by packing the real part of $X[N/2]$ into the imaginary part of $X[0]$.

Therefore, the functions in this library that produce the spectra of real signals (such as fft_bfp_forward_mono() and fft_bfp_forward_stereo()) will pack the spectra in a slightly less straight-forward manner (as compared with the complex DFTs):

X[f] $\longleftarrow X[f]$ for $1 \leq f < N/2$

X[0] $\longleftarrow X[0] + jX[N/2]$

where X is an $N/2$-element array of `complex_s32_t`.

Likewise, this is the encoding expected when computing the N-point inverse DFT, such as by fft_bfp_inverse_mono() or fft_bfp_inverse_stereo().

Note: One additional note, when performing a stereo DFT or inverse DFT, so as to preserve the in-place computation of the result, the spectra of the two signals will be encoded into adjacent blocks

of memory, with the second spectrum (i.e. associated with 'channel b') occupying the higher memory address.

4.11.4   Note: Library FFT Length Support

When computing DFTs this library relies on one or both of a pair of look-up tables which contain portions of the Discrete Fourier Transform matrix. Longer FFT lengths require larger look-up tables. When building using CMake, the maximum FFT length can be specified as a CMake option, and these tables are auto-generated at build time.

If not using CMake, you can manually generate these files using a python script included with the library. The script is located at `lib_xcore_math/scripts/gen_fft_table.py`. If generated manually, you must add the generated .c file as a source, and the directory containing `xmath_fft_lut.h` must be added as an include directory when compiling the library's files.

Note that the header file must be named `xmath_fft_lut.h` as it is included via `#include "xmath_fft_lut.h"`.

By default the tables contain the coefficients necessary to perform forward or inverse DFTs of up to 1024 points. If larger DFTs are required, or if the maximum required DFT size is known to be less than 1024 points, the `MAX_FFT_LEN_LOG2` CMake option can be modified from its default value of `10`.

The two look-up tables correspond to the decimation-in-time and decimation-in-frequency FFT algorithms, and the run-time symbols for the tables are `xmath_dit_fft_lut` and `xmath_dif_fft_lut` respectively. Each table contains $N - 4$ complex 32-bit values, with a size of $8 \cdot (N - 4)$ bytes each.

To manually regenerate the tables for a maximum FFT length of 16384 ($= 2^{14}$), supporting only the decimation-in-time algorithm, for example, use the following:

```
python lib_xcore_math/script/gen_fft_table.py --dit --max_fft_log2 14
```

Use the `--help` flag with `gen_fft_table.py` for a more detailed description of its syntax and parameters.

4.11.5   Note: Digital Filter Conversion

This library supports optimized implementations of 16- and 32-bit FIR filters, as well as cascaded 32-bit biquad filters. Each of these filter implementations requires that the filter coefficients be represented in a compatible form.

To assist with that, several python scripts are distributed with this library which can be used to convert existing floating-point filter coefficients into a code which is easily callable from within an xCore application.

Each script reads in floating-point filter coefficients from a file and computes a new representation for the filter with coefficients which attempt to maximize precision and are compatible with the `lib_xcore_math` filtering API.

Each script outputs two files which can be included in your own xCore application. The first output is a C source (`.c`) file containing the computed filter parameters and several function definitions for initializing and executing the generated filter. The second output is a C header (`.h`) file which can be `#included` into your own application to give access to those functions.

Additionally, each script also takes a user-provided filter name as an input parameter. The output files (as well as the function names within) include the filter name so that more than one filter can be generated and executed using this mechanism.

As an example, take the following command to generate a 32-bit FIR filter:

python lib_xcore_math/script/gen_fir_filter_s32.py MyFilter filter_coefs.txt

This command creates a filter named "MyFilter", with coefficients taken from a file `filter_coefs.txt`. Two output files will be generated, `MyFilter.c` and `MyFilter.h`. Including `MyFilter.h` provides access to 3 functions, `MyFilter_init()`, `MyFilter_add_sample()`, and `MyFilter()` which correspond to the library functions *filter_fir_s32_init()*, *filter_fir_s32_add_sample()* and *filter_fir_s32()* respectively.

Use the `--help` flag with the scripts for more detailed descriptions of inputs and other options.

Filter Type	Script
32-bit FIR	`lib_xcore_math/script/gen_fir_filter_s32.py`
16-bit FIR	`lib_xcore_math/script/gen_fir_filter_s16.py`
32-bit Biquad	`lib_xcore_math/script/gen_biquad_filter_s32.py`

5 Example Applications

Several example applications are offered to demonstrate use of the `lib_xcore_math` APIs through actual, simple code examples.

5.1 Building Examples

After configuring the CMake project (with the `-DDEV_LIB_XCORE_MATH=1` option), all the examples can be built by using the `make lib_xcore_math-examples` command within the build directory. Individual examples can be built using `make EXAMPLE_NAME`, where `EXAMPLE_NAME` is the example to build.

5.2 Running Examples

To run example `EXAMPLE_NAME` on the XCORE-AI-EXPLORER board, from the CMake build directory, run the following command (after building):

```
xrun --xscope example/EXAMPLE_NAME/EXAMPLE_NAME.xe
```

For instance, to run the `bfp_demo` example, use:

```
xrun --xscope example/bfp_demo/bfp_demo.xe
```

To run the example using the xCore simulator instead, use:

```
xsim example/EXAMPLE_NAME/EXAMPLE_NAME.xe
```

5.3 bfp_demo

The purpose of this application is to demonstrate, through example, how the arithmetic functions of `lib_xcore_math`'s block floating-point API may be used.

In it, three 32-bit BFP vectors are allocated, initialized and filled with random data. Then several BFP operations are applied using those vectors as inputs and/or outputs.

The example only demonstrates the real 32-bit arithmetic BFP functions (that is, functions with names `bfp_s32_*`). The real 16-bit (`bfp_s16_*`), complex 32-bit (`bfp_complex_s32_*`) and complex 16-bit (`bfp_complex_s16_*`) functions all use similar naming conventions.

5.4 vect_demo

The purpose of this application is to demonstrate, through example, how the arithmetic functions of `lib_xcore_math`'s lower-level vector API may be used.

In general the low-level arithmetic API are the functions in this library whose names begin with `vect_*`, such as *vect_s32_mul()* for element-wise multiplication of 32-bit vectors, and *vect_complex_s16_scale()* for multiplying a complex 16-bit vector by a complex scalar.

We assume that where the low-level API is being used it is because some behavior other than the default behavior of the high-level block floating-point API is required. Given that, rather than showcasing the breadth of operations available, this example examines first how to achieve comparable behavior to the BFP API, and then ways in which that behavior can be modified.

5.5 fft_demo

The purpose of this application is to demonstrate, through example, how the FFT functions of `lib_xcore_math`'s block floating-point API may be used.

In this example we demonstrate each of the offered forward and inverse FFTs of the BFP API.

5.6 filter_demo

The purpose of this application is to demonstrate, through example, how the functions of `lib_xcore_math`'s filtering vector API may be used.

The filtering API currently supports three different filter types:

- 32-bit FIR Filter
- 16-bit FIR Filter
- 32-bit Biquad Filter

This example app presents simple demonstrations of how to use each of these filter types.