CSI DSP Interface Manual

Release v4.2

Jan 18, 2022

平头哥半导体有限公司 T-HEAD Semiconductor Co.,LTD

地址: 杭州市余杭区向往街 1122 号欧美金融城 (EFC) 英国中心西楼 T6

邮编: 311121

网址: www.t-head.cn

Contents

CHAPTER 1

CSI DSP Software Library

1.1 Introduction

This user manual describes the CSI DSP software library, a suite of common signal processing functions for use on XT80x and XT90x processor based devices.

The library is divided into a number of functions each covering a specific category:

- Basic math functions
- Complex math functions
- Fast math functions
- Filters
- Interpolation functions
- Matrix functions
- Motor control functions
- Statistical functions
- Support functions
- Transforms

The library has separate functions for operating on 8-bit fixed-point, 16-bit fixed-point, 32-bit fixed-point and 32-bit floating-point values.

1.2 Using the Library

The library installer contains prebuilt versions of the libraries in the `lib` folder, for E906/E907, the library is:

- libcsi_xt900p32_math.a

The library functions are declared in the public file `csi_math.h` which is placed in the `include` folder. Simply include this file and link the appropriate library in the application and begin calling the library functions.

Note The old naming method with `csky_` start, will be obsoleted, the new naming method is started with `csi_`. The same functions for different kinds of processors will have the same name. And the method of started with `csi_` is recommended, the method of started with `csky_` will be obsoleted in the next version.

1.3 Different versions

Different versions of library for different ISA are shown blow:

versions	ISA	Optimization range
E804	CSKY DSP	Fixed-point functions
I805	CSKY VDSP2	Fixed-point functions
C860	CSKY VDSP2	All the functions
E906/E907	RISCV DSP	Fixed-point functions

Note Due to special requirement, individual functions are only realized for specific processors. The support of these functions for each version of library is as follows(Y for support, N for unsupport):

function name	E804	I805	C860	E906/E907
abs_max_q15	Y	Y	N	N
abs_max_q31	Y	Y	N	N
mult_rnd_q15	Y	Y	Y	N
dot_prod_u64xu8	N	Y	N	N
mult_q15xq31_sht	N	Y	N	N
sum_q15	N	Y	N	N
mat_inv_f32	Y	N	N	Y
mat_inv_f64	Y	N	N	Y
mat_mul_fast_q31	Y	N	N	Y
mat_mul_fast_q15	Y	N	N	Y
pow_int32	Y	Y	Y	N

1.4 Examples

CSI SDK has a number of examples which demonstrate how to use the library functions.

CHAPTER 2

Basic Math Functions

2.1 Vector Absolute Value

2.1.1 Functions

- *csi_abs_f32* : Floating-point vector absolute value.
- *csi_abs_q15* : Q15 vector absolute value.
- *csi_abs_q31* : Q31 vector absolute value.
- *csi_abs_q7* : Q7 vector absolute value.

2.1.2 Description

Computes the absolute value of a vector on an element-by-element basis.

```
pDst[n] = abs(pSrc[n]),   0 <= n < blockSize.
```

The functions support in-place computation allowing the source and destination pointers to reference the same memory buffer. There are separate functions for floating-point, Q7, Q15, and Q31 data types.

2.1.3 Function Docmentation

2.1.3.1 csi_abs_f32

```
void csi_abs_f32(float32_t *pSrc, float32_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to the input buffer

`*pDst`: points to the output buffer

`blockSize`: number of samples in each vector

Returns:

none

2.1.3.2 csi_abs_q15

```
void csi_abs_q15 (q15_t *pSrc, q15_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to the input buffer

`*pDst`: points to the output buffer

`blockSize`: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. The Q15 value -1 (0x8000) will be saturated to the maximum allowable positive value 0x7FFF.

2.1.3.3 csi_abs_q31

```
void csi_abs_q31 (q31_t *pSrc, q31_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to the input buffer

`*pDst`: points to the output buffer

`blockSize`: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. The Q31 value -1 (0x80000000) will be saturated to the maximum allowable positive value 0x7FFFFFFF.

2.1.3.4 csi_abs_q7

```
void csi_abs_q7 (q7_t *pSrc, q7_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to the input buffer

`*pDst`: points to the output buffer

`blockSize`: number of samples in each vector

Returns:

none

Conditions for optimum performance:

Input and output buffers should be aligned by 32-bit.

Scaling and Overflow Behavior:

The function uses saturating arithmetic. The Q7 value -1 (0x80) will be saturated to the maximum allowable positive value 0x7F.

2.2 Vector Absolute Max Value

2.2.1 Functions

- *csi_abs_max_q15* : Q15 vector absolute max value.
- *csi_abs_max_q31* : Q31 vector absolute max value.

2.2.2 Description

Computes the max absolute value of a vector on an element-by-element basis, which is only support E804/I805.

```
*pDst = max(abs(pSrc[n])),   0 <= n < blockSize.
```

The functions support in-place computation allowing the source and destination pointers to reference the same memory buffer. There are separate functions Q15 and Q31 data types.

2.2.3 Function Docmentation

2.2.3.1 csi_abs_max_q15

```
void csi_abs_max_q15 (q15_t *pSrc, q15_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to the input buffer

`*pDst`: points to the output buffer

`blockSize`: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. The Q15 value -1 (0x8000) will be saturated to the maximum allowable positive value 0x7FFF.

2.2.3.2 csi_abs_max_q31

```
void csi_abs_max_q31 (q31_t *pSrc, q31_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to the input buffer

`*pDst`: points to the output buffer

`blockSize`: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. The Q31 value -1 (0x80000000) will be saturated to the maximum allowable positive value 0x7FFFFFFF.

2.3 Vector Addition

2.3.1 Functions

- *csi_add_f32* : Floating-point vector addition.
- *csi_add_q15* : Q15 vector addition.
- *csi_add_q31* : Q31 vector addition.
- *csi_add_q7* : Q7 vector addition.

2.3.2 Description

Element-by-element addition of two vectors.

```
pDst[n] = pSrcA[n] + pSrcB[n],   0 <= n < blockSize.
```

There are separate functions for floating-point, Q7, Q15, and Q31 data types.

2.3.3 Function Docmentation

2.3.3.1 csi_add_f32

```
void csi_add_f32 (float32_t *pSrcA, float32_t *pSrcB, float32_t *pDst, uint32_t␣
↪blockSize)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`*pDst`: points to the output vector

`blockSize`: number of samples in each vector

Returns:

none

2.3.3.2 csi_add_q15

```
void csi_add_q15 (q15_t *pSrcA, q15_t *pSrcB, q15_t *pDst, uint32_t blockSize)
```

Parameters:

*pSrcA: points to the first input vector

*pSrcB: points to the second input vector

*pDst: points to the output vector

blockSize: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] will be saturated.

2.3.3.3 csi_add_q31

```
void csi_add_q31 (q31_t *pSrcA, q31_t *pSrcB, q31_t *pDst, uint32_t blockSize)
```

Parameters:

*pSrcA: points to the first input vector

*pSrcB: points to the second input vector

*pDst: points to the output vector

blockSize: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q31 range[0x80000000 0x7FFFFFFF] will be saturated.

2.3.3.4 csi_add_q7

```
void csi_add_q7 (q7_t *pSrcA, q7_t *pSrcB, q7_t *pDst, uint32_t blockSize)
```

Parameters:

*pSrcA: points to the first input vector

*pSrcB: points to the second input vector

*pDst: points to the output vector

blockSize: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q31 range[0x80 0x7F] will be saturated.

2.4 Vector Dot Product

2.4.1 Functions

- *csi_dot_prod_f32* : Dot product of floating-point vectors.
- *csi_dot_prod_q15* : Dot product of Q15 vectors.
- *csi_dot_prod_q31* : Dot product of Q31 vectors.
- *csi_dot_prod_q7* : Dot product of Q7 vectors.
- *csi_dot_prod_u64xu8* : Dot product of Uint64_t and Uint8_t vectors.

2.4.2 Description

Computes the dot product of two vectors. The vectors are multiplied element-by-element and then summed(dot_prod_u64xu8 only supports I805).

```
sum = pSrcA[0]*pSrcB[0] + pSrcA[1]*pSrcB[1] + ... + pSrcA[blockSize-
↪1]*pSrcB[blockSize-1]
```

There are separate functions for floating-point, Q7, Q15, and Q31 data types.

2.4.3 Function Docmentation

2.4.3.1 csi_dot_prod_f32

```
void csi_dot_prod_f32 (float32_t *pSrcA, float32_t *pSrcB, uint32_t blockSize,␣
↪float32_t *result)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`blockSize`: number of samples in each vector

`*result`: output result returned here

Returns:

none

2.4.3.2 csi_dot_prod_q15

```
void csi_dot_prod_q15 (q15_t *pSrcA, q15_t *pSrcB, uint32_t blockSize, q63_t *result)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`blockSize`: number of samples in each vector

`*result`: output result returned here

Returns:

none

Scaling and Overflow Behavior:

The intermediate multiplications are in 1.15 x 1.15 = 2.30 format and these results are added to a 64-bit accumulator in 34.30 format. Nonsaturating additions are used and given that there are 33 guard bits in the accumulator there is no risk of overflow. The return result is in 34.30 format.

2.4.3.3 csi_dot_prod_q31

```
void csi_dot_prod_q31 (q31_t *pSrcA, q31_t *pSrcB, uint32_t blockSize, q63_t *result)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`blockSize`: number of samples in each vector

`*result`: output result returned here

Returns:

none

Scaling and Overflow Behavior:

The intermediate multiplications are in 1.31 x 1.31 = 2.62 format and these are truncated to 2.48 format by discarding the lower 14 bits. The 2.48 result is then added without saturation to a 64-bit accumulator in 16.48 format. There are 15 guard bits in the accumulator and there is no risk of overflow as long as the length of the vectors is less than 2^16 elements. The return result is in 16.48 format.

2.4.3.4 csi_dot_prod_q7

```
void csi_dot_prod_q7 (q7_t *pSrcA, q7_t *pSrcB, uint32_t blockSize, q31_t *result)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`blockSize`: number of samples in each vector

`*result`: output result returned here

Returns:

none

Scaling and Overflow Behavior:

The intermediate multiplications are in 1.7 x 1.7 = 2.14 format and these results are added to an accumulator in 18.14 format. Nonsaturating additions are used and there is no danger of wrap around as long as the vectors are less than 2^18 elements long. The return result is in 18.14 format.

2.4.3.5 csi_dot_prod_u64xu8

```
void csi_dot_prod_u64xu8 (uint8_t *pSrcA, uint64_t *pSrcB, uint32_t blockSize, uint64_
↪t *result)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`blockSize`: number of samples in each vector

`*result`: output result returned here

Returns:

none

Scaling and Overflow Behavior:

A 64 bits accumulator is used in the intermediate multiply-accumulating. And there is no saturating arithmetic and guard bit. Be careful of overflow, when used.

2.5 Vector Multiplication

2.5.1 Functions

- *csi_mult_f32* : Floating-point vector multiplication.
- *csi_mult_q15* : Q15 vector multiplication.
- *csi_mult_q31* : Q31 vector multiplication.
- *csi_mult_q7* : Q7 vector multiplication.
- *csi_mult_rnd_q15* : Q15 vector multiplication with round.
- *csi_mult_q15xq31_sht* : Q15xQ31 vector multiplication with shift.

2.5.2 Description

Element-by-element multiplication of two vectors(csi_mult_rnd_q15 is not support for E906/E907, csi_mult_q15xq31_sht only supports I805).

```
pDst[n] = pSrcA[n] * pSrcB[n],   0 <= n < blockSize.
```

There are separate functions for floating-point, Q7, Q15, and Q31 data types.

2.5.3 Function Docmentation

2.5.3.1 csi_mult_f32

```
void csi_mult_f32 (float32_t *pSrcA, float32_t *pSrcB, float32_t *pDst, uint32_t␣
↪blockSize)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`*pDst`: points to the output vector

`blockSize`: number of samples in each vector

Returns:

none

2.5.3.2 csi_mult_q15

```
void csi_mult_q15 (q15_t *pSrcA, q15_t *pSrcB, q15_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`*pDst`: points to the output vector

`blockSize`: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] will be saturated.

2.5.3.3 csi_mult_q31

```
void csi_mult_q31 (q31_t *pSrcA, q31_t *pSrcB, q31_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`*pDst`: points to the output vector

`blockSize`: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q31 range[0x80000000 0x7FFFFFFF] will be saturated.

2.5.3.4 csi_mult_q7

```
void csi_mult_q7 (q7_t *pSrcA, q7_t *pSrcB, q7_t *pDst, uint32_t blockSize)
```

Parameters:

*pSrcA: points to the first input vector

*pSrcB: points to the second input vector

*pDst: points to the output vector

blockSize: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q7 range [0x80 0x7F] will be saturated.

2.5.3.5 csi_mult_rnd_q15

```
void csi_mult_rnd_q15 (q15_t *pSrcA, q15_t *pSrcB, q15_t *pDst, uint32_t blockSize)
```

Parameters:

*pSrcA: points to the first input vector

*pSrcB: points to the second input vector

*pDst: points to the output vector

blockSize: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] will be saturated. And before saturated, the results are rounded to positive infinity.

2.5.3.6 csi_mult_q15xq31_sht

```
void csi_mult_q15xq31_sht (q15_t *pSrcA, q31_t *pSrcB, uint32_t shiftValue, uint32_t␣
↪blockSize)
```

Parameters:

*pSrcA: points to the first input vector

*pSrcB: points to the second input vector and the output vector

shiftVale: number of right shift bitts of the output(shiftVale >= 0)

blockSize: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q31 range [0x80000000 0x7FFFFFFF] will be saturated. And before saturated, the results are right shifted shiftVale bits.

2.6 Vector Negate

2.6.1 Functions

- *csi_negate_f32* : Negates the elements of a floating-point vector.
- *csi_negate_q15* : Negates the elements of a Q15 vector.
- *csi_negate_q31* : Negates the elements of a Q31 vector.
- *csi_negate_q7* : Negates the elements of a Q7 vector.

2.6.2 Description

Negates the elements of a vector.

```
pDst[n] = -pSrc[n],   0 <= n < blockSize.
```

The functions support in-place computation allowing the source and destination pointers to reference the same memory buffer. There are separate functions for floating-point, Q7, Q15, and Q31 data types.

2.6.3 Function Docmentation

2.6.3.1 csi_negate_f32

```
void csi_negate_f32 (float32_t *pSrc, float32_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to the input vector

`*pDst`: points to the output vector

`blockSize`: number of samples in the vector

Returns:

none

2.6.3.2 csi_negate_q15

```
void csi_negate_q15 (q15_t *pSrc, q15_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to the input vector

`*pDst`: points to the output vector

`blockSize`: number of samples in the vector

Returns:

none

Conditions for optimum performance

Input and output buffers should be aligned by 32-bit

Scaling and Overflow Behavior:

The function uses saturating arithmetic. The Q15 value -1 (0x8000) will be saturated to the maximum allowable positive value 0x7FFF.

2.6.3.3 csi_negate_q31

```
void csi_negate_q31 (q31_t *pSrc, q31_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to the input vector

`*pDst`: points to the output vector

`blockSize`: number of samples in the vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. The Q31 value -1 (0x80000000) will be saturated to the maximum allowable positive value 0x7FFFFFFF.

2.6.3.4 csi_negate_q7

```
void csi_negate_q7 (q7_t *pSrc, q7_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to the input vector

`*pDst`: points to the output vector

`blockSize`: number of samples in the vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. The Q7 value -1 (0x80) will be saturated to the maximum allowable positive value 0x7F.

2.7 Vector Offset

2.7.1 Functions

- *csi_offset_f32* : Adds a constant offset to a floating-point vector.
- *csi_offset_q15* : Adds a constant offset to a Q15 vector.
- *csi_offset_q31* : Adds a constant offset to a Q31 vector.
- *csi_offset_q7* : Adds a constant offset to a Q7 vector.

2.7.2 Description

Adds a constant offset to each element of a vector.

```
pDst[n] = pSrc[n] + offset,   0 <= n < blockSize.
```

The functions support in-place computation allowing the source and destination pointers to reference the same memory buffer. There are separate functions for floating-point, Q7, Q15, and Q31 data types.

2.7.3 Function Docmentation

2.7.3.1 csi_offset_f32

```
void csi_offset_f32 (float32_t *pSrc, float32_t offset, float32_t *pDst, uint32_t␣
↪blockSize)
```

Parameters:

`*pSrc`: points to the input vector

`offset`: is the offset to be added

`*pDst`: points to the output vector

`blockSize`: number of samples in the vector

Returns:

none

2.7.3.2 csi_offset_q15

```
void csi_offset_q15 (q15_t *pSrc, q15_t offset, q15_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to the input vector

`offset`: is the offset to be added

`*pDst`: points to the output vector

`blockSize`: number of samples in the vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] are saturated.

2.7.3.3 csi_offset_q31

```
void csi_offset_q31 (q31_t *pSrc, q31_t offset, q31_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to the input vector

`offset`: is the offset to be added

`*pDst`: points to the output vector

`blockSize`: number of samples in the vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q31 range [0x80000000 0x7FFFFFFF] are saturated.

2.7.3.4 csi_offset_q7

```
void csi_offset_q7 (q7_t *pSrc, q7_t offset, q7_t *pDst, uint32_t blockSize)
```

Parameters:

*pSrc: points to the input vector

offset: is the offset to be added

*pDst: points to the output vector

blockSize: number of samples in the vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q31 range [0x80 0x7F] are saturated.

2.8 Vector Scale

2.8.1 Functions

- *csi_scale_f32* : Multiplies a floating-point vector by a scalar.
- *csi_scale_q15* : Multiplies a Q15 vector by a scalar.
- *csi_scale_q31* : Multiplies a Q31 vector by a scalar.
- *csi_scale_q7* : Multiplies a Q7 vector by a scalar.

2.8.2 Description

Multiply a vector by a scalar value. For floating-point data, the algorithm used is:

```
pDst[n] = pSrc[n] * scale,  0 <= n < blockSize.
```

In the fixed-point Q7, Q15, and Q31 functions, scale is represented by a fractional multiplication scaleFract and an arithmetic shift shift. The shift allows the gain of the scaling operation to exceed 1.0. The algorithm used with fixed-point data is:

```
pDst[n] = (pSrc[n] * scaleFract) << shift,  0 <= n < blockSize.
```

The overall scale factor applied to the fixed-point data is

```
scale = scaleFract * 2^shift.
```

The functions support in-place computation allowing the source and destination pointers to reference the same memory buffer.

2.8.3 Function Docmentation

2.8.3.1 csi_scale_f32

```
void csi_scale_f32 (float32_t *pSrc, float32_t scale, float32_t *pDst, uint32_t␣
↪blockSize)
```

Parameters:

`*pSrc`: points to the input vector

`scale`: scale factor to be applied

`*pDst`: points to the output vector

`blockSize`: number of samples in the vector

Returns:

none

2.8.3.2 csi_scale_q15

```
void csi_scale_q15 (q15_t *pSrc, q15_t scaleFract, int8_t shift, q15_t *pDst, uint32_
↪t blockSize)
```

Parameters:

`*pSrc`: points to the input vector

`scaleFract`: fractional portion of the scale value

`shift`: number of bits to shift the result by

`*pDst`: points to the output vector

`blockSize`: number of samples in the vector

Returns:

none

Scaling and Overflow Behavior:

The input data *pSrc and scaleFract are in 1.15 format. These are multiplied to yield a 2.30 intermediate result and this is shifted with saturation to 1.15 format.

2.8.3.3 csi_scale_q31

```
void csi_scale_q31 (q31_t *pSrc, q31_t scaleFract, int8_t shift, q31_t *pDst, uint32_
↪t blockSize)
```

Parameters:

`*pSrc`: points to the input vector

`scaleFract`: fractional portion of the scale value

`shift`: number of bits to shift the result by

`*pDst`: points to the output vector

`blockSize`: number of samples in the vector

Returns:

none

Scaling and Overflow Behavior:

The input data *pSrc and scaleFract are in 1.31 format. These are multiplied to yield a 2.62 intermediate result and this is shifted with saturation to 1.31 format.

2.8.3.4 csi_scale_q7

```
void csi_scale_q7 (q7_t *pSrc, q7_t scaleFract, int8_t shift, q7_t *pDst, uint32_t␣
↪blockSize)
```

Parameters:

`*pSrc`: points to the input vector

`scaleFract`: fractional portion of the scale value

`shift`: number of bits to shift the result by

`*pDst`: points to the output vector

`blockSize`: number of samples in the vector

Returns:

none

Scaling and Overflow Behavior:

The input data *pSrc and scaleFract are in 1.7 format. These are multiplied to yield a 2.14 intermediate result and this is shifted with saturation to 1.7 format.

2.9 Vector Shift

2.9.1 Functions

- *csi_shift_q15* : Shifts the elements of a Q15 vector a specified number of bits.
- *csi_shift_q31* : Shifts the elements of a Q31 vector a specified number of bits.
- *csi_shift_q7* : Shifts the elements of a Q7 vector a specified number of bits.

2.9.2 Description

Shifts the elements of a fixed-point vector by a specified number of bits. There are separate functions for Q7, Q15, and Q31 data types. The underlying algorithm used is:

```
pDst[n] = pSrc[n] << shift,   0 <= n < blockSize.
```

If shift is positive then the elements of the vector are shifted to the left. If shift is negative then the elements of the vector are shifted to the right.

The functions support in-place computation allowing the source and destination pointers to reference the same memory buffer.

2.9.3 Function Docmentation

2.9.3.1 csi_shift_q15

```
void csi_shift_q15 (q15_t *pSrc, int8_t shiftBits, q15_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to the input vector

`shiftBits`: number of bits to shift. A positive value shifts left; a negative value shifts right.

`*pDst`: points to the output vector

`blockSize`: number of samples in the vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] will be saturated.

2.9.3.2 csi_shift_q31

```
void csi_shift_q31 (q31_t *pSrc, int8_t shiftBits, q31_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to the input vector

`shiftBits`: number of bits to shift. A positive value shifts left; a negative value shifts right.

`*pDst`: points to the output vector

`blockSize`: number of samples in the vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q31 range [0x80000000 0x7FFFFFFF] will be saturated.

2.9.3.3 csi_shift_q7

```
void csi_shift_q7 (q7_t *pSrc, int8_t shiftBits, q7_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to the input vector

`shiftBits`: number of bits to shift. A positive value shifts left; a negative value shifts right.

`*pDst`: points to the output vector

`blockSize`: number of samples in the vector

Returns:

none

Conditions for optimum performance

Input and output buffers should be aligned by 32-bit

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q7 range [0x80 0x7F] will be saturated.

2.10 Vector Subtraction

2.10.1 Functions

- *csi_sub_f32* : Floating-point vector subtraction.
- *csi_sub_q15* : Q15 vector subtraction.
- *csi_sub_q31* : Q31 vector subtraction.
- *csi_sub_q7* : Q7 vector subtraction.

2.10.2 Description

Element-by-element subtraction of two vectors.

```
pDst[n] = pSrcA[n] - pSrcB[n],   0 <= n < blockSize.
```

There are separate functions for floating-point, Q7, Q15, and Q31 data types.

2.10.3 Function Docmentation

2.10.3.1 csi_sub_f32

```
void csi_sub_f32 (float32_t *pSrcA, float32_t *pSrcB, float32_t *pDst, uint32_t␣
↪blockSize)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`*pDst`: points to the output vector

`blockSize`: number of samples in each vector

Returns:

none

2.10.3.2 csi_sub_q15

```
void csi_sub_q15 (q15_t *pSrcA, q15_t *pSrcB, q15_t *pDst, uint32_t blockSize)
```

Parameters:

*pSrcA: points to the first input vector

*pSrcB: points to the second input vector

*pDst: points to the output vector

blockSize: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] will be saturated.

2.10.3.3 csi_sub_q31

```
void csi_sub_q31 (q31_t *pSrcA, q31_t *pSrcB, q31_t *pDst, uint32_t blockSize)
```

Parameters:

*pSrcA: points to the first input vector

*pSrcB: points to the second input vector

*pDst: points to the output vector

blockSize: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q31 range[0x80000000 0x7FFFFFFF] will be saturated.

2.10.3.4 csi_sub_q7

```
void csi_sub_q7 (q7_t *pSrcA, q7_t *pSrcB, q7_t *pDst, uint32_t blockSize)
```

Parameters:

*pSrcA: points to the first input vector

*pSrcB: points to the second input vector

*pDst: points to the output vector

blockSize: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q31 range[0x80 0x7F] will be saturated.

2.11 Vector Summation

2.11.1 Functions

- *csi_sum_q15* : Q15 vector sum.

2.11.2 Description

Sum all the elements in the vector, which only supports I805.

```
sum = pSrcA[0] + pSrcA[1] + ... + pSrcA[n],   0 <= n < blockSize.
```

There is only function for Q15 data types.

2.11.3 Function Docmentation

2.11.3.1 csi_sum_q15

```
void csi_sum_q15 (q15_t *pSrcA, q63_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrcA`: points to the input vector

`*pDst`: points to the output address

`blockSize`: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

A 64 bits accumulator is used in the function, there is 48 guard bits left. So, no saturating arithmetic is needed.

CHAPTER 3

Complex Math Functions

This set of functions operates on complex data vectors. The data in the complex arrays is stored in an interleaved fashion (real, imag, real, imag, ⋯). In the API functions, the number of samples in a complex array refers to the number of complex values; the array contains twice this number of real values.

3.1 Complex Conjugate

3.1.1 Functions

- *csi_cmplx_conj_f32* : Floating-point complex conjugate.
- *csi_cmplx_conj_q15* : Q15 complex conjugate.
- *csi_cmplx_conj_q31* : Q31 complex conjugate.

3.1.2 Description

Conjugates the elements of a complex data vector.

The pSrc points to the source data and pDst points to the where the result should be written. numSamples specifies the number of complex samples and the data in each array is stored in an interleaved fashion (real, imag, real, imag, ⋯). Each array has a total of 2*numSamples values. The underlying algorithm is used:

```
for (n=0; n<numSamples; n++) {
    pDst[(2*n)+0)] = pSrc[(2*n)+0];   // real part
```

(continues on next page)

(continued from previous page)

```
    pDst[(2*n)+1)] = -pSrc[(2*n)+1];    // imag part
}
```

There are separate functions for floating-point, Q15, and Q31 data types.

3.1.3 Function Docmentation

3.1.3.1 csi_cmplx_conj_f32

```
void csi_cmplx_conj_f32 (float32_t *pSrc, float32_t *pDst, uint32_t numSamples)
```

Parameters:

`*pSrc`: points to the input buffer

`*pDst`: points to the output buffer

`numSamples`: number of complex samples in each vector

Returns:

none

3.1.3.2 csi_cmplx_conj_q15

```
void csi_cmplx_conj_q15 (q15_t *pSrc, q15_t *pDst, uint32_t numSamples)
```

Parameters:

`*pSrc`: points to the input buffer

`*pDst`: points to the output buffer

`numSamples`: number of complex samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. The Q15 value -1 (0x8000) will be saturated to the maximum allowable positive value 0x7FFF.

3.1.3.3 csi_cmplx_conj_q31

```
void csi_cmplx_conj_q31 (q31_t *pSrc, q31_t *pDst, uint32_t numSamples)
```

Parameters:

`*pSrc`: points to the input buffer

`*pDst`: points to the output buffer

`numSamples`: number of complex samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. The Q31 value -1 (0x80000000) will be saturated to the maximum allowable positive value 0x7FFFFFFF.

3.2 Complex Dot Product

3.2.1 Functions

- *csi_cmplx_dot_prod_f32* : Floating-point complex dot product.
- *csi_cmplx_dot_prod_q15* : Q15 complex dot product.
- *csi_cmplx_dot_prod_q31* : Q31 complex dot product.

3.2.2 Description

Computes the dot product of two complex vectors. The vectors are multiplied element-by-element and then summed.

The pSrcA points to the first complex input vector and pSrcB points to the second complex input vector. numSamples specifies the number of complex samples and the data in each array is stored in an interleaved fashion (real, imag, real, imag, ⋯). Each array has a total of 2*numSamples values.

The underlying algorithm is used:

```
realResult=0;
imagResult=0;
for (n=0; n<numSamples; n++) {
    realResult += pSrcA[(2*n)+0]*pSrcB[(2*n)+0] - pSrcA[(2*n)+1]*pSrcB[(2*n)+1];
    imagResult += pSrcA[(2*n)+0]*pSrcB[(2*n)+1] + pSrcA[(2*n)+1]*pSrcB[(2*n)+0];
}
```

There are separate functions for floating-point, Q15, and Q31 data types.

3.2.3 Function Docmentation

3.2.3.1 csi_cmplx_dot_prod_f32

```
void csi_cmplx_dot_prod_f32 (float32_t *pSrcA, float32_t *pSrcB, uint32_t numSamples,␣
↪float32_t *realResult, float32_t *imagResult)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`numSamples`: number of complex samples in each vector

`*realResult`: real part of the result returned here

`*imagResult`: imaginary part of the result returned here

Returns:

none

3.2.3.2 csi_cmplx_dot_prod_q15

```
void csi_cmplx_dot_prod_q15 (q15_t *pSrcA, q15_t *pSrcB, uint32_t numSamples, q31_t␣
↪*realResult, q31_t *imagResult)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`numSamples`: number of complex samples in each vector

`*realResult`: real part of the result returned here

`*imagResult`: imaginary part of the result returned here

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The intermediate 1.15 by 1.15 multiplications are performed with full precision and yield a 2.30 result. These are accumulated in a 64-bit accumulator with 34.30 precision. As a final step, the accumulators are converted to 8.24 format. The return results realResult and imagResult are in 8.24 format.

3.2.3.3 csi_cmplx_dot_prod_q31

```
void  csi_cmplx_dot_prod_q31 (q31_t *pSrcA, q31_t *pSrcB, uint32_t numSamples, q63_t␣
↪*realResult, q63_t *imagResult)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`numSamples`: number of complex samples in each vector

`*realResult`: real part of the result returned here

`*imagResult`: imaginary part of the result returned here

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The intermediate 1.31 by 1.31 multiplications are performed with 64-bit precision and then shifted to 16.48 format. The internal real and imaginary accumulators are in 16.48 format and provide 15 guard bits. Additions are nonsaturating and no overflow will occur as long as numSamples is less than 32768. The return results realResult and imagResult are in 16.48 format. Input down scaling is not required.

3.3 Complex Magnitude

3.3.1 Functions

- *csi_cmplx_mag_f32* : Floating-point complex magnitude.
- *csi_cmplx_mag_q15* : Q15 complex magnitude.
- *csi_cmplx_mag_q31* : Q31 complex magnitude.

3.3.2 Description

Computes the magnitude of the elements of a complex data vector.

The pSrc points to the source data and pDst points to the where the result should be written. numSamples specifies the number of complex samples in the input array and the data is stored in an interleaved fashion (real, imag, real, imag, ⋯). The input array has a total of 2*numSamples values; the output array has a total of numSamples values. The underlying algorithm is used:

```
for(n=0; n<numSamples; n++) {
    pDst[n] = sqrt(pSrc[(2*n)+0]^2 + pSrc[(2*n)+1]^2);
}
```

There are separate functions for floating-point, Q15, and Q31 data types.

3.3.3 Function Docmentation

3.3.3.1 csi_cmplx_mag_f32

```
void csi_cmplx_mag_f32 (float32_t *pSrc, float32_t *pDst, uint32_t numSamples)
```

Parameters:

`*pSrc`: points to complex input buffer

`*pDst`: points to real output buffer

`numSamples`: number of complex samples in each vector

Returns:

none

3.3.3.2 csi_cmplx_mag_q15

```
void csi_cmplx_mag_q15 (q15_t *pSrc, q15_t *pDst, uint32_t numSamples)
```

Parameters:

`*pSrc`: points to complex input buffer

`*pDst`: points to real output buffer

`numSamples`: number of complex samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function implements 1.15 by 1.15 multiplications and finally output is converted into 2.14 format. If imag and real are 0x8000, implementation may overflow.

3.3.3.3 csi_cmplx_mag_q31

```
void csi_cmplx_mag_q31 (q31_t *pSrc, q31_t *pDst, uint32_t numSamples)
```

Parameters:

`*pSrc`: points to complex input buffer

`*pDst`: points to real output buffer

`numSamples`: number of complex samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function implements 1.31 by 1.31 multiplications and finally output is converted into 2.30 format. Input down scaling is not required.

3.4 Complex Magnitude Squared

3.4.1 Functions

- *csi_cmplx_mag_squared_f32* : Floating-point complex magnitude squared.
- *csi_cmplx_mag_squared_q15* : Q15 complex magnitude squared.
- *csi_cmplx_mag_squared_q31* : Q31 complex magnitude squared.

3.4.2 Description

Computes the magnitude squared of the elements of a complex data vector.

The pSrc points to the source data and pDst points to the where the result should be written. numSamples specifies the number of complex samples in the input array and the data is stored in an interleaved fashion (real, imag, real, imag, ⋯). The input array has a total of 2*numSamples values; the output array has a total of numSamples values.

The underlying algorithm is used:

```
for (n=0; n<numSamples; n++) {
    pDst[n] = pSrc[(2*n)+0]^2 + pSrc[(2*n)+1]^2;
}
```

There are separate functions for floating-point, Q15, and Q31 data types.

3.4.3 Function Docmentation

3.4.3.1 csi_cmplx_mag_squared_f32

```
void csi_cmplx_mag_squared_f32 (float32_t *pSrc, float32_t *pDst, uint32_t numSamples)
```

Parameters:

`*pSrc`: points to complex input buffer

`*pDst`: points to real output buffer

`numSamples`: number of complex samples in each vector

Returns:

none

3.4.3.2 csi_cmplx_mag_squared_q15

```
void csi_cmplx_mag_squared_q15 (q15_t *pSrc, q15_t *pDst, uint32_t numSamples)
```

Parameters:

`*pSrc`: points to complex input buffer

`*pDst`: points to real output buffer

`numSamples`: number of complex samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function implements 1.15 by 1.15 multiplications and finally output is converted into 3.13 format.

3.4.3.3 csi_cmplx_mag_squared_q31

```
void csi_cmplx_mag_squared_q31 (q31_t *pSrc, q31_t *pDst, uint32_t numSamples)
```

Parameters:

`*pSrc`: points to complex input buffer

`*pDst`: points to real output buffer

`numSamples`: number of complex samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function implements 1.31 by 1.31 multiplications and finally output is converted into 3.29 format. Input down scaling is not required.

3.5 Complex-by-Complex Multiplication

3.5.1 Functions

- *csi_cmplx_mult_cmplx_f32* : Floating-point complex-by-complex multiplication.
- *csi_cmplx_mult_cmplx_q15* : Q15 complex-by-complex multiplication.
- *csi_cmplx_mult_cmplx_q31* : Q31 complex-by-complex multiplication.

3.5.2 Description

Multiplies a complex vector by another complex vector and generates a complex result. The data in the complex arrays is stored in an interleaved fashion (real, imag, real, imag, ⋯). The parameter numSamples represents the number of complex samples processed. The complex arrays have a total of 2*numSamples real values.

The underlying algorithm is used:

```
for(n=0; n<numSamples; n++) {
    pDst[(2*n)+0] = pSrcA[(2*n)+0] * pSrcB[(2*n)+0] - pSrcA[(2*n)+1] * pSrcB[(2*n)+1];
    pDst[(2*n)+1] = pSrcA[(2*n)+0] * pSrcB[(2*n)+1] + pSrcA[(2*n)+1] * pSrcB[(2*n)+0];
}
```

There are separate functions for floating-point, Q15, and Q31 data types.

3.5.3 Function Docmentation

3.5.3.1 csi_cmplx_mult_cmplx_f32

```
void csi_cmplx_mult_cmplx_f32 (float32_t *pSrcA, float32_t *pSrcB, float32_t *pDst,␣
↪uint32_t numSamples)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`*pDst`: points to the output vector

`numSamples`: number of complex samples in each vector

Returns:

none

3.5.3.2 csi_cmplx_mult_cmplx_q15

```
void csi_cmplx_mult_cmplx_q15 (q15_t *pSrcA, q15_t *pSrcB, q15_t *pDst, uint32_t␣
↪numSamples)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`*pDst`: points to the output vector

`numSamples`: number of complex samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function implements 1.15 by 1.15 multiplications and finally output is converted into 3.13 format.

3.5.3.3 csi_cmplx_mult_cmplx_q31

```
void csi_cmplx_mult_cmplx_q31 (q31_t *pSrcA, q31_t *pSrcB, q31_t *pDst, uint32_t␣
↪numSamples)
```

Parameters:

`*pSrcA`: points to the first input vector

`*pSrcB`: points to the second input vector

`*pDst`: points to the output vector

`numSamples`: number of complex samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function implements 1.31 by 1.31 multiplications and finally output is converted into 3.29 format. Input down scaling is not required.

3.6 Complex-by-Real Multiplication

3.6.1 Functions

- *csi_cmplx_mult_real_f32* : Floating-point complex-by-real multiplication.
- *csi_cmplx_mult_real_q15* : Q15 complex-by-real multiplication.
- *csi_cmplx_mult_real_q31* : Q31 complex-by-real multiplication.

3.6.2 Description

Multiplies a complex vector by a real vector and generates a complex result. The data in the complex arrays is stored in an interleaved fashion (real, imag, real, imag, ⋯). The parameter numSamples represents the number of complex samples processed. The complex arrays have a total of 2*numSamples real values while the real array has a total of numSamples real values.

The underlying algorithm is used:

```
for (n=0; n<numSamples; n++) {
    pCmplxDst[(2*n)+0] = pSrcCmplx[(2*n)+0] * pSrcReal[n];
    pCmplxDst[(2*n)+1] = pSrcCmplx[(2*n)+1] * pSrcReal[n];
}
```

There are separate functions for floating-point, Q15, and Q31 data types.

3.6.3 Function Docmentation

3.6.3.1 csi_cmplx_mult_real_f32

```
void csi_cmplx_mult_real_f32 (float32_t *pSrcCmplx, float32_t *pSrcReal, float32_t␣
↪*pCmplxDst, uint32_t numSamples)
```

Parameters:

`*pSrcCmplx`: points to the complex input vector

`*pSrcReal`: points to the real input vector

`*pCmplxDst`: points to the complex output vector

`numSamples`: number of samples in each vector

Returns:

none

3.6.3.2 csi_cmplx_mult_real_q15

```
void csi_cmplx_mult_real_q15 (q15_t *pSrcCmplx, q15_t *pSrcReal, q15_t *pCmplxDst,␣
↪uint32_t numSamples)
```

Parameters:

`*pSrcCmplx`: points to the complex input vector

`*pSrcReal`: points to the real input vector

`*pCmplxDst`: points to the complex output vector

`numSamples`: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] will be saturated.

3.6.3.3 csi_cmplx_mult_real_q31

```
void csi_cmplx_mult_real_q31 (q31_t *pSrcCmplx, q31_t *pSrcReal, q31_t *pCmplxDst,␣
↪uint32_t numSamples)
```

Parameters:

`*pSrcCmplx`: points to the complex input vector

`*pSrcReal`: points to the real input vector

`*pCmplxDst`: points to the complex output vector

`numSamples`: number of samples in each vector

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q31 range[0x80000000 0x7FFFFFFF] will be saturated.

CHAPTER 4

Controller Functions

4.1 PID Motor Control

4.1.1 Functions

- *csi_pid_f32* : Processing function for the floating-point PID Control.
- *csi_pid_q31* : Processing function for the Q15 PID Control.
- *csi_pid_q15* : Processing function for the Q31 PID Control.
- *csi_pid_init_f32* : Initialization function for the floating-point PID Control.
- *csi_pid_init_q31* : Initialization function for the Q15 PID Control.
- *csi_pid_init_q15* : Initialization function for the Q31 PID Control.
- *csi_pid_reset_f32* : Reset function for the floating-point PID Control.
- *csi_pid_reset_q31* : Reset function for the Q15 PID Control.
- *csi_pid_reset_q15* : Reset function for the Q31 PID Control.

4.1.2 Description

A Proportional Integral Derivative (PID) controller is a generic feedback control loop mechanism widely used in industrial control systems. A PID controller is the most commonly used type of feedback controller.

This set of functions implements (PID) controllers for Q15, Q31, and floating-point data types. The functions operate on a single sample of data and each call to the function returns a single processed value. S points to an instance of the PID control data structure. in is the input sample value. The functions return the output value.

Algorithm:

```
y[n] = y[n-1] + A0 * x[n] + A1 * x[n-1] + A2 * x[n-2]
A0 = Kp + Ki + Kd
A1 = (-Kp ) - (2 * Kd )
A2 = Kd
```

where Kp is proportional constant, Ki is Integral constant and Kd is Derivative constant

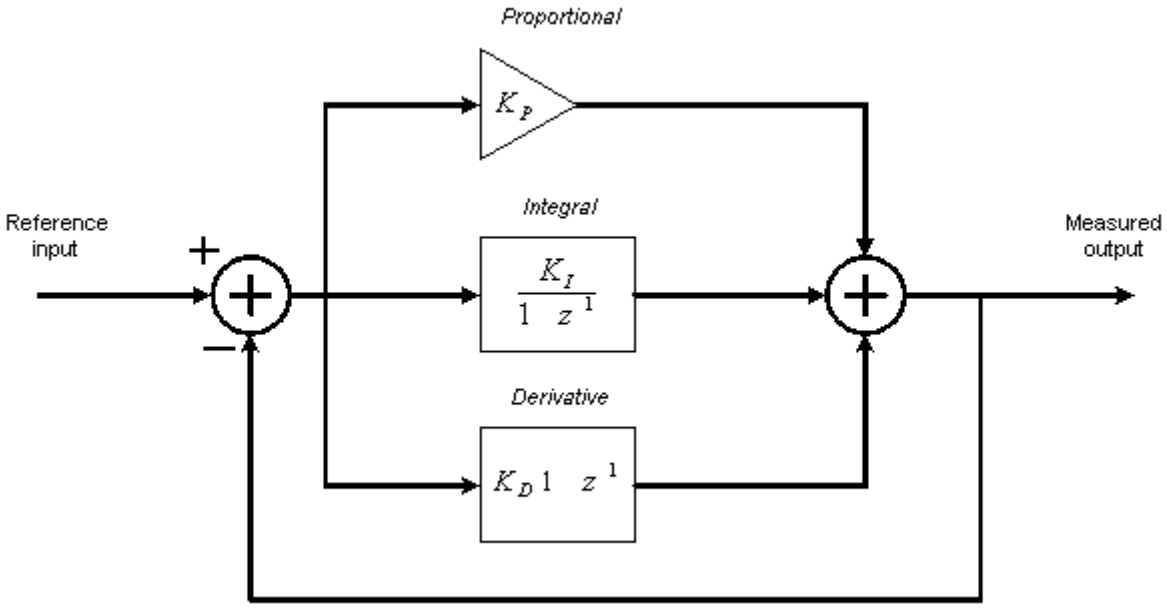


Figure4.1: Proportional Integral Derivative Controller

The PID controller calculates an "error" value as the difference between the measured output and the reference input. The controller attempts to minimize the error by adjusting the process control inputs. The proportional value determines the reaction to the current error, the integral value determines the reaction based on the sum of recent errors, and the derivative value determines the reaction based on the rate at which the error has been changing.

Instance Structure

The Gains A0, A1, A2 and state variables for a PID controller are stored together in an instance data structure. A separate instance structure must be defined for each PID Controller. There are separate instance structure declarations for each of the 3 supported data types.

Reset Functions

There is also an associated reset function for each data type which clears the state array.

Initialization Functions

There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Initializes the Gains A0, A1, A2 from Kp,Ki, Kd gains.
- Zeros out the values in the state buffer.

Instance structure cannot be placed into a const data section and it is recommended to use the initialization function.

Fixed-Point Behavior

Care must be taken when using the fixed-point versions of the PID Controller functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

4.1.3 Function Docmentation

4.1.3.1 csi_pid_f32

```
float32_t csi_pid_f32 (csi_pid_instance_f32 *S, float32_t in)
```

Parameters:

`*S`: points to an instance of the PID structure.

`in`: input sample to process

Returns:

out processed output sample.

4.1.3.2 csi_pid_q31

```
q31_t csi_pid_q31 (csi_pid_instance_q31 *S, q31_t in)
```

Parameters:

`*S`: points to an instance of the PID structure.

`in`: input sample to process

Returns:

out processed output sample.

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clip. In order to avoid overflows completely the input signal must be scaled down by 2 bits as there are four additions. After all multiply-accumulates are performed, the 2.62 accumulator is truncated to 1.32 format and then saturated to 1.31 format.

4.1.3.3 csi_pid_q15

```
q15_t csi_pid_q15 (csi_pid_instance_q15 *S, q15_t in)
```

Parameters:

`*S`: points to an instance of the PID structure.

`in`: input sample to process

Returns:

out processed output sample.

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. Both Gains and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. After all additions have been performed, the accumulator is truncated to 34.15 format by discarding low 15 bits. Lastly, the accumulator is saturated to yield a result in 1.15 format.

4.1.3.4 csi_pid_init_f32

```
void csi_pid_init_f32 (csi_pid_instance_f32 *S, int32_t resetStateFlag)
```

Parameters:

`*S`: points to an instance of the PID structure.

`resetStateFlag`: flag to reset the state. 0 = no change in state & 1 = reset the state.

Returns:

none.

Description:

The resetStateFlag specifies whether to set state to zero or not.

The function computes the structure fields: A0, A1 A2 using the proportional gain(Kp), integral gain(Ki) and derivative gain(Kd) also sets the state variables to all zeros.

4.1.3.5 csi_pid_init_q31

```
void csi_pid_init_q31 (csi_pid_instance_q31 *S, int32_t resetStateFlag)
```

Parameters:

`*S`: points to an instance of the PID structure.

`resetStateFlag`: flag to reset the state. 0 = no change in state & 1 = reset the state.

Returns:

none.

Description:

The resetStateFlag specifies whether to set state to zero or not.

The function computes the structure fields: A0, A1 A2 using the proportional gain(Kp), integral gain(Ki) and derivative gain(Kd) also sets the state variables to all zeros.

4.1.3.6 csi_pid_init_q15

```
void csi_pid_init_q15 (csi_pid_instance_q15 *S, int32_t resetStateFlag)
```

Parameters:

`*S`: points to an instance of the PID structure.

`resetStateFlag`: flag to reset the state. 0 = no change in state & 1 = reset the state.

Returns:

none.

Description:

The resetStateFlag specifies whether to set state to zero or not.

The function computes the structure fields: A0, A1 A2 using the proportional gain(Kp), integral gain(Ki) and derivative gain(Kd) also sets the state variables to all zeros.

4.1.3.7 csi_pid_reset_f32

```
void csi_pid_reset_f32 (csi_pid_instance_f32 *S)
```

Parameters:

`*S`: points to an instance of the PID structure.

Returns:

none.

Description:

The function resets the state buffer to zeros.

4.1.3.8 csi_pid_reset_q31

```
void csi_pid_reset_q31 (csi_pid_instance_q31 *S)
```

Parameters:

`*S`: points to an instance of the PID structure.

Returns:

none.

Description:

The function resets the state buffer to zeros.

4.1.3.9 csi_pid_reset_q15

```
void csi_pid_reset_q15 (csi_pid_instance_q15 *S)
```

Parameters:

`*S`: points to an instance of the PID structure.

Returns:

none.

Description:

The function resets the state buffer to zeros.

4.2 Vector Clarke Transform

4.2.1 Functions

- *csi_clarke_f32* : Floating-point Clarke transform.
- *csi_clarke_q31* : Clarke transform for Q31 version.

4.2.2 Description

Forward Clarke transform converts the instantaneous stator phases into a two-coordinate time invariant vector. Generally the Clarke transform uses three-phase currents Ia, Ib and Ic to calculate currents in the two-phase orthogonal stator axis Ialpha and Ibeta. When Ialpha is superposed with Ia as shown in the figure below

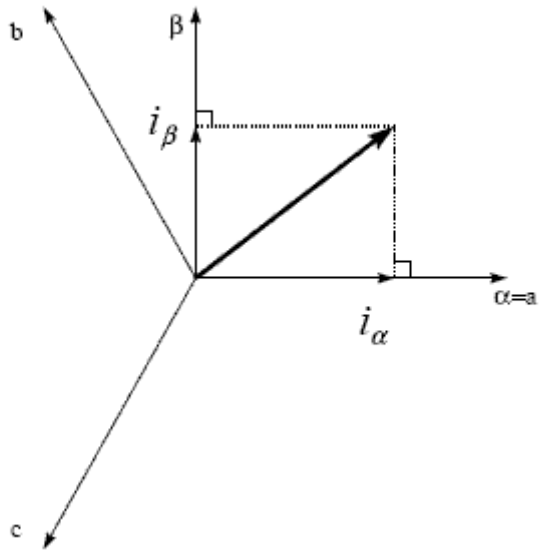


Figure4.2: Stator current space vector and its components in (a,b).

and Ia + Ib + Ic = 0, in this condition Ialpha and Ibeta can be calculated using only Ia and Ib.

The function operates on a single sample of data and each call to the function returns the processed output. The library provides separate functions for Q31 and floating-point data types.

Algorithm

$$\text{pIalpha} = \text{Ia}$$
$$\text{pIbeta} = (1/\sqrt{3})\,\text{Ia} + (2/\sqrt{3})\,\text{Ib}$$

where Ia and Ib are the instantaneous stator phases and pIalpha and pIbeta are the two coordinates of time invariant vector.

Fixed-Point Behavior

Care must be taken when using the Q31 version of the Clarke transform. In particular, the overflow and saturation behavior of the accumulator used must be considered. Refer to the function specific documentation below for usage guidelines.

4.2.3 Function Docmentation

4.2.3.1 csi_clarke_f32

```
void csi_clarke_f32 (float32_t Ia, float32_t Ib, float32_t *pIalpha, float32_t␣
↪*pIbeta)
```

Parameters:

`Ia`: input three-phase coordinate a.

`Ib`: input three-phase coordinate b.

`*pIalpha`: points to output two-phase orthogonal vector axis alpha.

`*pIbeta`: points to output two-phase orthogonal vector axis beta.

Returns:

none.

4.2.3.2 csi_clarke_q31

```
void csi_clarke_q31 (q31_t Ia, q31_t Ib, q31_t *pIalpha, q31_t *pIbeta)
```

Parameters:

`Ia`: input three-phase coordinate a.

`Ib`: input three-phase coordinate b.

`*pIalpha`: points to output two-phase orthogonal vector axis alpha.

`*pIbeta`: points to output two-phase orthogonal vector axis beta.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using an internal 32-bit accumulator. The accumulator maintains 1.31 format by truncating lower 31 bits of the intermediate multiplication in 2.62 format. There is saturation on the addition, hence there is no risk of overflow.

4.3 Vector Inverse Clarke Transform

4.3.1 Functions

- *csi_inv_clarke_f32* : Floating-point Inverse Clarke transform.
- *csi_inv_clarke_q31* : Inverse Clarke transform for Q31 version.

4.3.2 Description

Inverse Clarke transform converts the two-coordinate time invariant vector into instantaneous stator phases.

The function operates on a single sample of data and each call to the function returns the processed output. The library provides separate functions for Q31 and floating-point data types.

Algorithm

$$pIa = Ialpha$$
$$pIb = (-1/2)\,Ialpha + (\sqrt{3}/2)\,Ibeta$$

where pIa and pIb are the instantaneous stator phases and Ialpha and Ibeta are the two coordinates of time invariant vector.

Fixed-Point Behavior

Care must be taken when using the Q31 version of the Clarke transform. In particular, the overflow and saturation behavior of the accumulator used must be considered. Refer to the function specific documentation below for usage guidelines.

4.3.3 Function Docmentation

4.3.3.1 csi_inv_clarke_f32

```
void csi_inv_clarke_f32 (float32_t Ialpha, float32_t Ibeta, float32_t *pIa, float32_t␣
↪*pIb)
```

Parameters:

`Ialpha`: input two-phase orthogonal vector axis alpha.

`Ibeta`: input output two-phase orthogonal vector axis beta.

`*pIa`: points to output three-phase coordinate a.

`*pIb`: points to output three-phase coordinate b.

Returns:

none.

4.3.3.2 csi_inv_clarke_q31

```
void csi_inv_clarke_q31 (q31_t Ialpha, q31_t Ibeta, q31_t *pIa, q31_t *pIb)
```

Parameters:

`Ialpha`: input two-phase orthogonal vector axis alpha.

`Ibeta`: input two-phase orthogonal vector axis beta.

`*pIa`: points to output three-phase coordinate a.

`*pIb`: points to output three-phase coordinate b.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using an internal 32-bit accumulator. The accumulator maintains 1.31 format by truncating lower 31 bits of the intermediate multiplication in 2.62 format. There is saturation on the subtraction, hence there is no risk of overflow.

4.4 Vector Park Transform

4.4.1 Functions

- *csi_park_f32* : Floating-point Park transform.
- *csi_park_q31* : Park transform for Q31 version.

4.4.2 Description

Forward Park transform converts the input two-coordinate vector to flux and torque components. The Park transform can be used to realize the transformation of the Ialpha and the Ibeta currents from the stationary to the moving reference frame and control the spatial relationship between the stator vector current and rotor flux vector. If we consider the d axis aligned with the rotor flux, the diagram below shows the current vector and the relationship from the two reference frames:

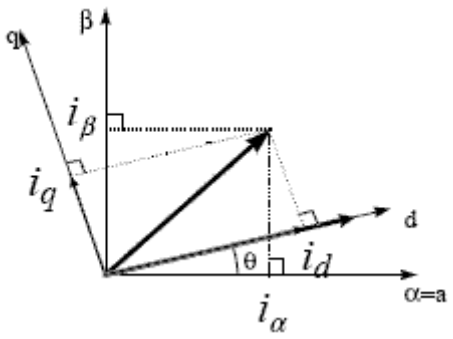


Figure4.3: Stator current space vector and its component in (a,b) and in the d,q rotating reference frame

The function operates on a single sample of data and each call to the function returns the processed output. The library provides separate functions for Q31 and floating-point data types.

Algorithm

$$pId = Ialpha * cosVal + Ibeta * sinVal$$
$$pIq = - Ialpha\ sinVal + Ibeta * cosVal$$

where Ialpha and Ibeta are the stator vector components, pId and pIq are rotor vector components and cosVal and sinVal are the cosine and sine values of theta (rotor flux position).

Fixed-Point Behavior

Care must be taken when using the Q31 version of the Park transform. In particular, the overflow and saturation behavior of the accumulator used must be considered. Refer to the function specific documentation below for usage guidelines.

4.4.3 Function Docmentation

4.4.3.1 csi_park_f32

```
void csi_park_f32 (float32_t Ialpha, float32_t Ibeta, float32_t *pId, float32_t *pIq,␣
↪float32_t sinVal, float32_t cosVal)
```

Parameters:

`Ialpha`: input two-phase vector coordinate alpha.

`Ibeta`: input two-phase vector coordinate beta.

`*pId`: points to output rotor reference frame d.

`*pIq`: points to output rotor reference frame q.

`sinVal`: sine value of rotation angle theta.

`cosVal`: cosine value of rotation angle theta.

Returns:

none.

4.4.3.2 csi_park_q31

```
void csi_park_q31 (q31_t Ialpha, q31_t Ibeta, q31_t *pId, q31_t *pIq, q31_t sinVal,␣
↪q31_t cosVal)
```

Parameters:

`Ialpha`: input two-phase vector coordinate alpha.

`Ibeta`: input two-phase vector coordinate beta.

`*pId`: points to output rotor reference frame d.

`*pIq`: points to output rotor reference frame q.

`sinVal`: sine value of rotation angle theta.

`cosVal`: cosine value of rotation angle theta.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using an internal 32-bit accumulator. The accumulator maintains 1.31 format by truncating lower 31 bits of the intermediate multiplication in 2.62 format. There is saturation on the addition and subtraction, hence there is no risk of overflow.

4.5 Vector Inverse Park transform

4.5.1 Functions

- *csi_inv_park_f32* : Floating-point Inverse Park transform.
- *csi_inv_park_q31* : Inverse Park transform for Q31 version.

4.5.2 Description

Inverse Park transform converts the input flux and torque components to two-coordinate vector.

The function operates on a single sample of data and each call to the function returns the processed output. The library provides separate functions for Q31 and floating-point data types.

Algorithm

$$pIalpha = Id * cosVal - Iq * sinVal$$
$$pIbeta = Id * sinVal + Iq * cosVal$$

where pIalpha and pIbeta are the stator vector components, Id and Iq are rotor vector components and cosVal and sinVal are the cosine and sine values of theta (rotor flux position).

Fixed-Point Behavior

Care must be taken when using the Q31 version of the Park transform. In particular, the overflow and saturation behavior of the accumulator used must be considered. Refer to the function specific documentation below for usage guidelines.

4.5.3 Function Docmentation

4.5.3.1 csi_inv_park_f32

```
void csi_inv_park_f32 (float32_t Id, float32_t Iq, float32_t *pIalpha, float32_t␣
↪*pIbeta, float32_t sinVal, float32_t cosVal)
```

Parameters:

`Id`: input coordinate of rotor reference frame d.

`Iq`: input coordinate of rotor reference frame q.

`*pIalpha`: points to output two-phase orthogonal vector axis alpha.

`*pIbeta`: points to output two-phase orthogonal vector axis beta.

`sinVal`: sine value of rotation angle theta.

`cosVal`: cosine value of rotation angle theta.

Returns:

none.

4.5.3.2 csi_inv_park_q31

```
void csi_inv_park_q31 (q31_t Id, q31_t Iq, q31_t *pIalpha, q31_t *pIbeta, q31_t␣
↪sinVal, q31_t cosVal)
```

Parameters:

`Id`: input coordinate of rotor reference frame d.

`Iq`: input coordinate of rotor reference frame q.

`*pIalpha`: points to output two-phase orthogonal vector axis alpha.

`*pIbeta`: points to output two-phase orthogonal vector axis beta.

`sinVal`: sine value of rotation angle theta.

`cosVal`: cosine value of rotation angle theta.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using an internal 32-bit accumulator. The accumulator maintains 1.31 format by truncating lower 31 bits of the intermediate multiplication in 2.62 format. There is saturation on the addition, hence there is no risk of overflow.

4.6 Sine Cosine

4.6.1 Functions

- *csi_sin_cos_f32* : Floating-point sin_cos function.
- *csi_sin_cos_q31* : Q31 sin_cos function.

4.6.2 Description

Computes the trigonometric sine and cosine values using a combination of table lookup and linear interpolation. There are separate functions for Q31 and floating-point data types. The input to the floating-point version is in degrees while the fixed-point Q31 have a scaled input with the range [-1 0.9999] mapping to [-180 +180] degrees.

The floating point function also allows values that are out of the usual range. When this happens, the function will take extra time to adjust the input value to the range of [-180 180].

The implementation is based on table lookup using 360 values together with linear interpolation. The steps used are:

1. Calculation of the nearest integer table index.
2. Compute the fractional portion (fract) of the input.
3. Fetch the value corresponding to index from sine table to y0 and also value from index+1 to y1.
4. Sine value is computed as psinVal = y0 + (fract * (y1 - y0)).
5. Fetch the value corresponding to index from cosine table to y0 and also value from index+1 to y1.
6. Cosine value is computed as pcosVal = y0 + (fract * (y1 - y0)).

4.6.3 Function Docmentation

4.6.3.1 csi_sin_cos_f32

```
void csi_sin_cos_f32 (float32_t theta, float32_t *pSinVal, float32_t *pCosVal)
```

Parameters:

`theta`: input value in degrees.

`*pSinVal`: points to the processed sine output.

`*pCosVal`: points to the processed cos output.

Returns:

none.

4.6.3.2 csi_sin_cos_q31

```
void csi_sin_cos_q31 (q31_t theta, q31_t *pSinVal, q31_t *pCosVal)
```

Parameters:

`theta`: input value in degrees.

`*pSinVal`: points to the processed sine output.

`*pCosVal`: points to the processed cos output.

Returns:

none.

note:

The Q31 input value is in the range [-1 0.999999] and is mapped to a degree value in the range [-180 179].

CHAPTER 5

Fast Math Functions

This set of functions provides a fast approximation to sine, cosine, and square root. As compared to most of the other functions in the CSI math library, the fast math functions operate on individual values and not arrays. There are separate functions for Q15, Q31, and floating-point data.

5.1 Cosine

5.1.1 Functions

- *csi_cos_f32* : Fast approximation to the trigonometric cosine function for floating-point data.
- *csi_cos_q15* : Fast approximation to the trigonometric cosine function for Q15 data.
- *csi_cos_q31* : Fast approximation to the trigonometric cosine function for Q31 data.

5.1.2 Description

Computes the trigonometric cosine function using a combination of table lookup and linear interpolation. There are separate functions for Q15, Q31, and floating-point data types. The input to the floating-point version is in radians while the fixed-point Q15 and Q31 have a scaled input with the range [0 +0.9999] mapping to [0 2*pi). The fixed-point range is chosen so that a value of 2*pi wraps around to 0.

The implementation is based on table lookup using 256 values together with linear interpolation. The steps used are:

1. Calculation of the nearest integer table index
2. Compute the fractional portion (fract) of the table index.

3. The final result equals (1.0f-fract)*a + fract*b;

where

```
b=Table[index+0];
c=Table[index+1];
```

5.1.3 Function Docmentation

5.1.3.1 csi_cos_f32

```
float32_t csi_cos_f32 (float32_t x)
```

Parameters:

`x`: input value in radians.

Returns:

cos(x).

5.1.3.2 csi_cos_q15

```
q15_t csi_cos_q15 (q15_t x)
```

Parameters:

`x`: input value in radians.

Returns:

cos(x).

Scaling and Overflow Behavior:

The Q15 input value is in the range [0 +0.9999] and is mapped to a radian value in the range [0 2*pi).

5.1.3.3 csi_cos_q31

```
q31_t csi_cos_q31 (q31_t x)
```

Parameters:

`x`: input value in radians.

Returns:

cos(x).

Scaling and Overflow Behavior:

The Q31 input value is in the range [0 +0.9999] and is mapped to a radian value in the range [0 2*pi).

5.2 Sine

5.2.1 Functions

- *csi_sin_f32* : Fast approximation to the trigonometric sine function for floating-point data.
- *csi_sin_q15* : Fast approximation to the trigonometric sine function for Q15 data.
- *csi_sin_q31* : Fast approximation to the trigonometric sine function for Q31 data.

5.2.2 Description

Computes the trigonometric sine function using a combination of table lookup and linear interpolation. There are separate functions for Q15, Q31, and floating-point data types. The input to the floating-point version is in radians while the fixed-point Q15 and Q31 have a scaled input with the range [0 +0.9999] mapping to [0 2*pi). The fixed-point range is chosen so that a value of 2*pi wraps around to 0.

The implementation is based on table lookup using 256 values together with linear interpolation. The steps used are:

1. Calculation of the nearest integer table index
2. Compute the fractional portion (fract) of the table index.
3. The final result equals (1.0f-fract)*a + fract*b;

where

```
b=Table[index+0];
c=Table[index+1];
```

5.2.3 Function Docmentation

5.2.3.1 csi_sin_f32

```
float32_t csi_sin_f32 (float32_t x)
```

Parameters:

`x`: input value in radians.

Returns:

sin(x).

5.2.3.2 csi_sin_q15

```
q15_t csi_sin_q15 (q15_t x)
```

Parameters:

`x`: input value in radians.

Returns:

sin(x).

Scaling and Overflow Behavior:

The Q15 input value is in the range [0 +0.9999] and is mapped to a radian value in the range [0 2*pi).

5.2.3.3 csi_sin_q31

```
q31_t csi_sin_q31 (q31_t x)
```

Parameters:

`x`: input value in radians.

Returns:

sin(x).

Scaling and Overflow Behavior:

The Q31 input value is in the range [0 +0.9999] and is mapped to a radian value in the range [0 2*pi).

5.3 Square Root

5.3.1 Functions

- *csi_sqrt_q15* : Q15 square root function.
- *csi_sqrt_q31* : Q31 square root function.

5.3.2 Description

Computes the square root of a number. There are separate functions for Q15 and Q31 data types. When a CPU with FPU is used, the instruction fsqrts is used to compute the result, while a CPU without FPU, Newton-Raphson algorithm is used. For Newton-Raphson algorithm, this is an iterative algorithm of the form:

```
x1 = x0 - f(x0)/f'(x0)
```

where x1 is the current estimate, x0 is the previous estimate, and f’(x0) is the derivative of f() evaluated at x0. For the square root function, the algorithm reduces to:

```
x0 = in/2          [initial guess]
x1 = 1/2 * ( x0 + in / x0)       [each iteration]
```

5.3.3 Function Docmentation

5.3.3.1 csi_sqrt_q15

```
csi_status csi_sqrt_q15 (q15_t in, q15_t *pOut)
```

Parameters:

`in`: input value. `pOut`: square root of input value.

Returns:

The function returns CSKY_MATH_SUCCESS if input value is positive value or CSKY_MATH_ARGUMENT_ERROR if in is negative value and returns zero output for negative values.

Scaling and Overflow Behavior:

The Q15 input value is in the range [0 +0.9999] and is mapped to a radian value in the range [0 2*pi).

5.3.3.2 csi_sqrt_q31

```
csi_status csi_sqrt_q31 (q31_t in, q31_t *pOut)
```

Parameters:

`in`: input value. `pOut`: square root of input value.

Returns:

The function returns CSKY_MATH_SUCCESS if input value is positive value or CSKY_MATH_ARGUMENT_ERROR if in is negative value and returns zero output for negative values.

Note

When the hard float instruction fsqrts is used, the accuracy will be lost from 3 LSB to 7 LSB. So as the functions, who call this function.

CHAPTER 6

Filtering Functions

6.1 Biquad Cascade IIR Filters Using Direct Form I Structure

6.1.1 Functions

- *csi_biquad_cascade_df1_f32* : Processing function for the floating-point Biquad cascade filter.
- *csi_biquad_cascade_df1_q15* : Processing function for the Q15 Biquad cascade filter.
- *csi_biquad_cascade_df1_q31* : Processing function for the Q31 Biquad cascade filter.
- *csi_biquad_cascade_df1_fast_q15* : Processing function for the Q15 Biquad cascade filter.
- *csi_biquad_cascade_df1_fast_q31* : Processing function for the Q31 Biquad cascade filter.
- *csi_biquad_cascade_df1_init_f32* : Initialization function for the floating-point Biquad cascade filter.
- *csi_biquad_cascade_df1_init_q15* : Initialization function for the Q15 Biquad cascade filter.
- *csi_biquad_cascade_df1_init_q31* : Initialization function for the Q31 Biquad cascade filter.

6.1.2 Description

This set of functions implements arbitrary order recursive (IIR) filters. The filters are implemented as a cascade of second order Biquad sections. The functions support Q15, Q31 and floating-point data types.

The functions operate on blocks of input and output data and each call to the function processes blockSize samples through the filter. pSrc points to the array of input data and pDst points to the array of output data. Both arrays contain blockSize values.

Algorithm

Each Biquad stage implements a second order filter using the difference equation:

```
y[n] = b0 * x[n] + b1 * x[n-1] + b2 * x[n-2] + a1 * y[n-1] + a2 * y[n-2]
```

A Direct Form I algorithm is used with 5 coefficients and 4 state variables per stage.

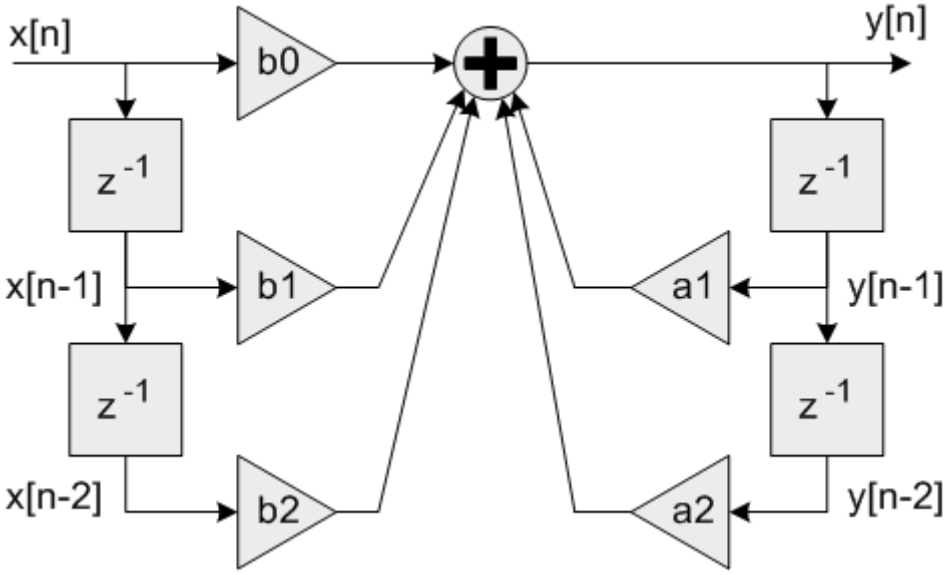


Figure6.1: Single Biquad filter stage

Coefficients b0, b1 and b2 multiply the input signal x[n] and are referred to as the feedforward coefficients. Coefficients a1 and a2 multiply the output signal y[n] and are referred to as the feedback coefficients. Pay careful attention to the sign of the feedback coefficients. Some design tools use the difference equation

```
y[n] = b0 * x[n] + b1 * x[n-1] + b2 * x[n-2] - a1 * y[n-1] - a2 * y[n-2]
```

In this case the feedback coefficients a1 and a2 must be negated when used with the CSI DSP Library.

Higher order filters are realized as a cascade of second order sections. numStages refers to the number of second order stages used. For example, an 8th order filter would be realized with numStages=4 second order stages.

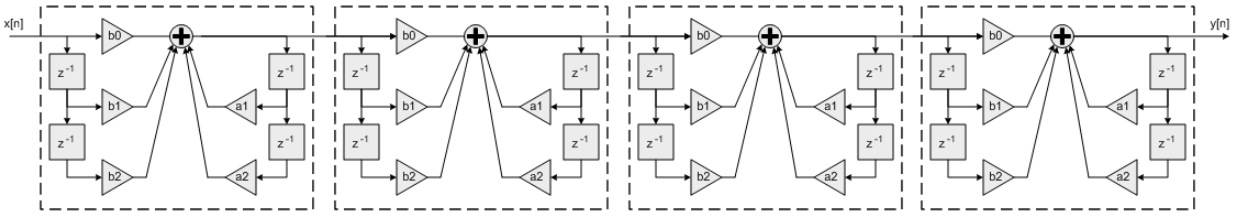


Figure6.2: 8th order filter using a cascade of Biquad stages

A 9th order filter would be realized with numStages=5 second order stages with the coefficients for one of the stages configured as a first order filter (b2=0 and a2=0).

The pState points to state variables array. Each Biquad stage has 4 state variables x[n-1], x[n-2], y[n-1], and y[n-2]. The state variables are arranged in the pState array as:

```
{x[n-1], x[n-2], y[n-1], y[n-2]}
```

The 4 state variables for stage 1 are first, then the 4 state variables for stage 2, and so on. The state array has a total length of 4*numStages values. The state variables are updated after each block of data is processed, the coefficients are untouched.

Instance Structure

The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient arrays may be shared among several instances while state variable arrays cannot be shared. There are separate instance structure declarations for each of the 3 supported data types.

Init Functions

There is also an associated initialization function for each data type. The initialization function performs following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numStages, pCoeffs, pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros before static initialization. The code below statically initializes each of the 3 different data type filter instance structures

```
csi_biquad_casd_df1_inst_f32 S1 = {numStages, pState, pCoeffs};
csi_biquad_casd_df1_inst_q15 S2 = {numStages, pState, pCoeffs, postShift};
csi_biquad_casd_df1_inst_q31 S3 = {numStages, pState, pCoeffs, postShift};
```

where numStages is the number of Biquad stages in the filter; pState is the address of the state buffer; pCoeffs is the address of the coefficient buffer; postShift shift to be applied.

Fixed-Point Behavior

Care must be taken when using the Q15 and Q31 versions of the Biquad Cascade filter functions. Following issues must be considered:

- Scaling of coefficients
- Filter gain
- Overflow and saturation

Scaling of coefficients: Filter coefficients are represented as fractional values and coefficients are restricted to lie in the range [-1 +1). The fixed-point functions have an additional scaling parameter postShift which allow the filter coefficients to exceed the range [+1 -1). At the output of the filter ' s accumulator is a shift register which shifts the result by postShift bits.

This essentially scales the filter coefficients by 2^postShift. For example, to realize the coefficients

```
{1.5, -0.8, 1.2, 1.6, -0.9}
```

set the pCoeffs array to:

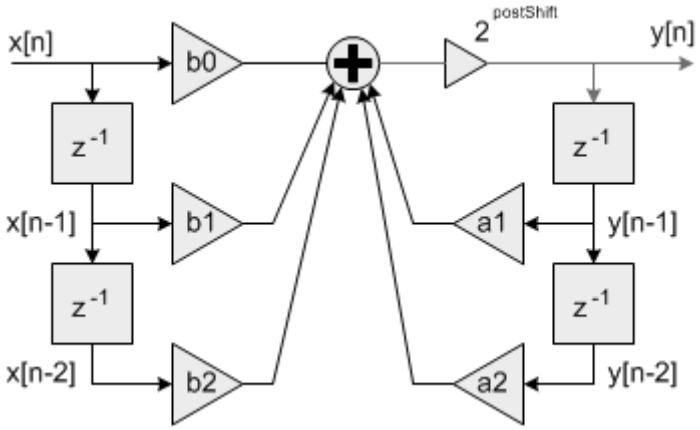


Figure6.3: Fixed-point Biquad with shift by postShift bits after accumulator

```
{0.75, -0.4, 0.6, 0.8, -0.45}
```

and set postShift=1

Filter gain: The frequency response of a Biquad filter is a function of its coefficients. It is possible for the gain through the filter to exceed 1.0 meaning that the filter increases the amplitude of certain frequencies. This means that an input signal with amplitude < 1.0 may result in an output > 1.0 and these are saturated or overflowed based on the implementation of the filter. To avoid this behavior the filter needs to be scaled down such that its peak gain < 1.0 or the input signal must be scaled down so that the combination of input and filter are never overflowed.

Overflow and saturation: For Q15 and Q31 versions, it is described separately as part of the function specific documentation below.

6.1.3 Function Docmentation

6.1.3.1 csi_biquad_cascade_df1_f32

```
void csi_biquad_cascade_df1_f32 (const csi_biquad_casd_df1_inst_f32 *S, float32_t␣
↪*pSrc, float32_t *pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the floating-point Biquad cascade structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

6.1.3.2 csi_biquad_cascade_df1_q15

```
void csi_biquad_cascade_df1_q15 (const csi_biquad_casd_df1_inst_q15 *S, q15_t *pSrc,␣
↪q15_t *pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the floating-point Biquad cascade structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. The accumulator is then shifted by postShift bits to truncate the result to 1.15 format by discarding the low 16 bits. Finally, the result is saturated to 1.15 format.

6.1.3.3 csi_biquad_cascade_df1_q31

```
void csi_biquad_cascade_df1_q31 (const csi_biquad_casd_df1_inst_q31 *S, q31_t *pSrc,␣
↪q31_t *pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the floating-point Biquad cascade structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clip. In order to avoid overflows completely the input signal must be scaled down by 2 bits and lie in the range [-0.25 +0.25). After all 5 multiply-accumulates are performed, the 2.62 accumulator is shifted by postShift bits and the result truncated to 1.31 format by discarding the low 32 bits.

6.1.3.4 csi_biquad_cascade_df1_fast_q15

```
void csi_biquad_cascade_df1_fast_q15 (const csi_biquad_casd_df1_inst_q15 *S, q15_t␣
↪*pSrc, q15_t *pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the floating-point Biquad cascade structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

Scaling and Overflow Behavior:

This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around and distorts the result. In order to avoid overflows completely the input signal must be scaled down by two bits and lie in the range [-0.25 +0.25). The 2.30 accumulator is then shifted by postShift bits and the result truncated to 1.15 format by discarding the low 16 bits.

Refer to the function *csi_biquad_cascade_df1_q15()* for a slower implementation of this function which uses 64-bit accumulation to avoid wrap around distortion. Both the slow and the fast versions use the same instance structure. Use the function *csi_biquad_cascade_df1_init_q15()* to initialize the filter structure.

6.1.3.5 csi_biquad_cascade_df1_fast_q31

```
void csi_biquad_cascade_df1_fast_q31 (const csi_biquad_casd_df1_inst_q31 *S, q31_t␣
↪*pSrc, q31_t *pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the floating-point Biquad cascade structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

Scaling and Overflow Behavior:

This function is optimized for speed at the expense of fixed-point precision and overflow protection. The result of each 1.31 x 1.31 multiplication is truncated to 2.30 format. These intermediate results are added to a 2.30 accumulator. Finally, the accumulator is saturated and converted to a 1.31 result. The fast version has the same overflow behavior as the standard version and provides less precision since it discards the low 32 bits of each multiplication result. In order to avoid overflows completely the input signal must be scaled down by two bits and lie in the range [-0.25 +0.25). Use the intialization function *csi_biquad_cascade_df1_init_q31()* to initialize filter structure.

Refer to the function *csi_biquad_cascade_df1_q31()* for a slower implementation of this function which uses 64-bit accumulation to provide higher precision. Both the slow and the fast versions use the same instance structure. Use the function *csi_biquad_cascade_df1_init_q31()* to initialize the filter structure.

6.1.3.6 csi_biquad_cascade_df1_init_f32

```
void csi_biquad_cascade_df1_init_f32 (csi_biquad_casd_df1_inst_f32 *S, uint8_t␣
↪numStages, float32_t *pCoeffs, float32_t *pState)
```

Parameters:

`*S`: points to an instance of the floating-point Biquad cascade structure.

`numStages`: number of 2nd order stages in the filter.

`*pCoeffs`: points to the filter coefficients array.

`*pState`: points to the state array.

Returns:

none

Coefficient and State Ordering:

The coefficients are stored in the array pCoeffs in the following order:

```
{b10, b11, b12, a11, a12, b20, b21, b22, a21, a22, ...}
```

where b1x and a1x are the coefficients for the first stage, b2x and a2x are the coefficients for the second stage, and so on. The pCoeffs array contains a total of 5*numStages values.

The pState is a pointer to state array. Each Biquad stage has 4 state variables x[n-1], x[n-2], y[n-1], and y[n-2]. The state variables are arranged in the pState array as:

```
{x[n-1], x[n-2], y[n-1], y[n-2]}
```

The 4 state variables for stage 1 are first, then the 4 state variables for stage 2, and so on. The state array has a total length of 4*numStages values. The state variables are updated after each block of data is processed; the coefficients are untouched.

6.1.3.7 csi_biquad_cascade_df1_init_q15

```
void csi_biquad_cascade_df1_init_q15 (csi_biquad_casd_df1_inst_q15 *S, uint8_t␣
↪numStages, q15_t *pCoeffs, q15_t *pState, int8_t postShift)
```

Parameters:

`*S`: points to an instance of the floating-point Biquad cascade structure.

`numStages`: number of 2nd order stages in the filter.

`*pCoeffs`: points to the filter coefficients array.

`*pState`: points to the state array.

`postShift` Shift to be applied to the accumulator result. Varies according to the coefficients format

Returns:

none

Coefficient and State Ordering:

The coefficients are stored in the array pCoeffs in the following order:

```
{b10, 0, b11, b12, a11, a12, b20, 0, b21, b22, a21, a22, ...}
```

where b1x and a1x are the coefficients for the first stage, b2x and a2x are the coefficients for the second stage, and so on. The pCoeffs array contains a total of 6*numStages values.

The state variables are stored in the array pState. Each Biquad stage has 4 state variables x[n-1], x[n-2], y[n-1], and y[n-2]. The state variables are arranged in the pState array as:

```
{x[n-1], x[n-2], y[n-1], y[n-2]}
```

The 4 state variables for stage 1 are first, then the 4 state variables for stage 2, and so on. The state array has a total length of 4*numStages values. The state variables are updated after each block of data is processed; the coefficients are untouched.

6.1.3.8 csi_biquad_cascade_df1_init_q31

```
void csi_biquad_cascade_df1_init_q31 (csi_biquad_casd_df1_inst_q31 *S, uint8_t␣
↪numStages, q31_t *pCoeffs, q31_t *pState, int8_t postShift)
```

Parameters:

`*S`: points to an instance of the floating-point Biquad cascade structure.

`numStages`: number of 2nd order stages in the filter.

`*pCoeffs`: points to the filter coefficients array.

`*pState`: points to the state array.

`postShift` Shift to be applied to the accumulator result. Varies according to the coefficients format

Returns:

none

Coefficient and State Ordering:

The coefficients are stored in the array pCoeffs in the following order:

```
{b10, b11, b12, a11, a12, b20, b21, b22, a21, a22, ...}
```

where b1x and a1x are the coefficients for the first stage, b2x and a2x are the coefficients for the second stage, and so on. The pCoeffs array contains a total of 5*numStages values.

The pState points to state variables array. Each Biquad stage has 4 state variables x[n-1], x[n-2], y[n-1], and y[n-2]. The state variables are arranged in the pState array as:

```
{x[n-1], x[n-2], y[n-1], y[n-2]}
```

The 4 state variables for stage 1 are first, then the 4 state variables for stage 2, and so on. The state array has a total length of 4*numStages values. The state variables are updated after each block of data is processed; the coefficients are untouched.

6.2 Biquad Cascade IIR Filters Using a Direct Form II Transposed Structure

6.2.1 Functions

- *csi_biquad_cascade_df2T_f32* : Processing function for the floating-point transposed direct form II Biquad cascade filter.
- *csi_biquad_cascade_df2T_init_f32* : Initialization function for the floating-point transposed direct form II Biquad cascade filter.
- *csi_biquad_cascade_stereo_df2T_f32* : Processing function for the floating-point transposed direct form II Biquad cascade filter.
- *csi_biquad_cascade_stereo_df2T_init_f32* : Initialization function for the floating-point transposed direct form II Biquad cascade filter.

6.2.2 Description

This set of functions implements arbitrary order recursive (IIR) filters using a transposed direct form II structure. The filters are implemented as a cascade of second order Biquad sections. These functions provide a slight memory savings as compared to the direct form I Biquad filter functions. Only floating-point data is supported.

This function operate on blocks of input and output data and each call to the function processes blockSize samples through the filter. pSrc points to the array of input data and pDst points to the array of output data. Both arrays contain blockSize values.

Algorithm

Each Biquad stage implements a second order filter using the difference equation:

```
y[n] = b0 * x[n] + d1
d1 = b1 * x[n] + a1 * y[n] + d2
d2 = b2 * x[n] + a2 * y[n]
```

where d1 and d2 represent the two state values.

A Biquad filter using a transposed Direct Form II structure is shown below.

Coefficients b0, b1, and b2 multiply the input signal x[n] and are referred to as the feedforward coefficients. Coefficients a1 and a2 multiply the output signal y[n] and are referred to as the feedback coefficients. Pay careful attention to the sign of the feedback coefficients. Some design tools flip the sign of the feedback coefficients:

```
y[n] = b0 * x[n] + d1;
d1 = b1 * x[n] - a1 * y[n] + d2;
d2 = b2 * x[n] - a2 * y[n];
```

In this case the feedback coefficients a1 and a2 must be negated when used with the CSI DSP Library.

Higher order filters are realized as a cascade of second order sections. numStages refers to the number of second order stages used. For example, an 8th order filter would be realized with numStages=4 second order stages. A 9th order filter would be realized with numStages=5 second order stages with the coefficients for one of the stages configured as a first order filter (b2=0 and a2=0).

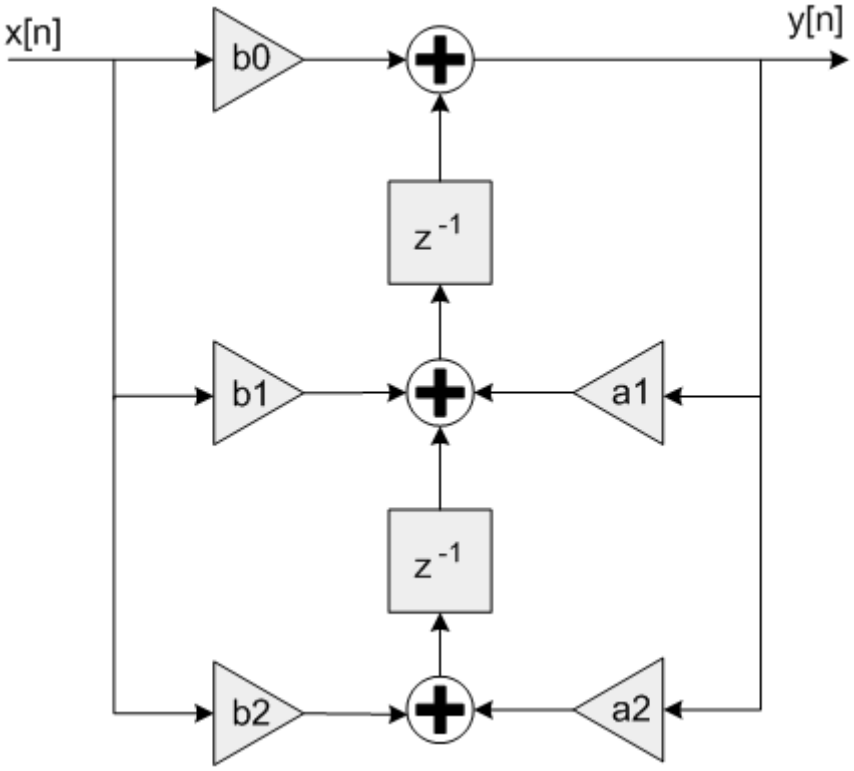


Figure6.4: Single transposed Direct Form II Biquad

pState points to the state variable array. Each Biquad stage has 2 state variables d1 and d2. The state variables are arranged in the pState array as:

```
{d11, d12, d21, d22, ...}
```

where d1x refers to the state variables for the first Biquad and d2x refers to the state variables for the second Biquad. The state array has a total length of 2*numStages values. The state variables are updated after each block of data is processed; the coefficients are untouched.

The CSI library contains Biquad filters in both Direct Form I and transposed Direct Form II. The advantage of the Direct Form I structure is that it is numerically more robust for fixed-point data types. That is why the Direct Form I structure supports Q15 and Q31 data types. The transposed Direct Form II structure, on the other hand, requires a wide dynamic range for the state variables d1 and d2. Because of this, the CSI library only has a floating-point version of the Direct Form II Biquad. The advantage of the Direct Form II Biquad is that it requires half the number of state variables, 2 rather than 4, per Biquad stage.

Instance Structure

The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient arrays may be shared among several instances while state variable arrays cannot be shared.

Init Functions

There is also an associated initialization function. The initialization function performs following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numStages, pCoeffs, pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros before static initialization. For example, to statically initialize the instance structure use

```
csi_biquad_cascade_df2T_instance_f32 S1 = {numStages, pState, pCoeffs};
```

where numStages is the number of Biquad stages in the filter; pState is the address of the state buffer. pCoeffs is the address of the coefficient buffer;

6.2.3 Function Docmentation

6.2.3.1 csi_biquad_cascade_df2T_f32

```
void csi_biquad_cascade_df2T_f32 (const csi_biquad_cascade_df2T_instance_f32 *S,␣
↪float32_t *pSrc, float32_t *pDst, uint32_t blockSize)
```

Parameters:

*S: points to an instance of the filter data structure.

*pSrc: points to the block of input data.

*pDst: points to the block of output data.

blockSize: number of samples to process.

Returns:

none

6.2.3.2 csi_biquad_cascade_df2T_init_f32

```
void csi_biquad_cascade_df2T_init_f32 (csi_biquad_cascade_df2T_instance_f32 *S, uint8_
↪t numStages, float32_t *pCoeffs, float32_t *pState)
```

Parameters:

*S: points to an instance of the filter data structure.

numStages: number of 2nd order stages in the filter.

*pCoeffs: points to the filter coefficients.

*pState: points to the state buffer.

Returns:

none

Coefficient and State Ordering:

The coefficients are stored in the array pCoeffs in the following order:

```
{b10, b11, b12, a11, a12, b20, b21, b22, a21, a22, ...}
```

where b1x and a1x are the coefficients for the first stage, b2x and a2x are the coefficients for the second stage, and so on. The pCoeffs array contains a total of 5*numStages values.

The pState is a pointer to state array. Each Biquad stage has 2 state variables d1, and d2. The 2 state variables for stage 1 are first, then the 2 state variables for stage 2, and so on. The state array has a total length of 2*numStages values. The state variables are updated after each block of data is processed; the coefficients are untouched.

6.2.3.3 csi_biquad_cascade_stereo_df2T_f32

```
void csi_biquad_cascade_stereo_df2T_f32 (const csi_biquad_cascade_stereo_df2T_
↪instance_f32 *S, float32_t *pSrc, float32_t *pDst, uint32_t blockSize)
```

Parameters:

*S: points to an instance of the filter data structure.

*pSrc: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process.

Returns:

none

6.2.3.4 csi_biquad_cascade_stereo_df2T_init_f32

```
void csi_biquad_cascade_stereo_df2T_init_f32 (csi_biquad_cascade_stereo_df2T_instance_
↪f32 *S, uint8_t numStages, float32_t *pCoeffs, float32_t *pState)
```

Parameters:

`*S`: points to an instance of the filter data structure.

`numStages`: number of 2nd order stages in the filter.

`*pCoeffs`: points to the filter coefficients.

`*pState`: points to the state buffer.

Returns:

none

Coefficient and State Ordering:

The coefficients are stored in the array pCoeffs in the following order:

```
{b10, b11, b12, a11, a12, b20, b21, b22, a21, a22, ...}
```

where b1x and a1x are the coefficients for the first stage, b2x and a2x are the coefficients for the second stage, and so on. The pCoeffs array contains a total of 5*numStages values.

The pState is a pointer to state array. Each Biquad stage has 2 state variables d1, and d2 for each channel. The 2 state variables for stage 1 are first, then the 2 state variables for stage 2, and so on. The state array has a total length of 2*numStages values. The state variables are updated after each block of data is processed; the coefficients are untouched.

6.3 Convolution

6.3.1 Functions

- *csi_conv_f32* : Convolution of floating-point sequences.
- *csi_conv_q31* : Convolution of Q31 sequences.
- *csi_conv_q15* : Convolution of Q15 sequences.
- *csi_conv_q7* : Convolution of Q7 sequences.
- *csi_conv_fast_q31* : Convolution of Q31 sequences.
- *csi_conv_fast_q15* : Convolution of Q15 sequences.
- *csi_conv_fast_opt_q15* : Convolution of Q15 sequences.
- *csi_conv_opt_q15* : Convolution of Q15 sequences.
- *csi_conv_opt_q7* : Convolution of Q7 sequences.

6.3.2 Description

Convolution is a mathematical operation that operates on two finite length vectors to generate a finite length output vector. Convolution is similar to correlation and is frequently used in filtering and data analysis. The CSI DSP library contains functions for convolving Q7, Q15, Q31, and floating-point data types.

Algorithm

Let a[n] and b[n] be sequences of length srcALen and srcBLen samples respectively. Then the convolution

```
c[n] = a[n] * b[n]
```

is defined as

$$c[n] = \sum_{k=0}^{srcALen} a[k]b[n-k]$$

Note that c[n] is of length srcALen + srcBLen - 1 and is defined over the interval n=0, 1, 2, …, srcALen + srcBLen - 2. pSrcA points to the first input vector of length srcALen and pSrcB points to the second input vector of length srcBLen. The output result is written to pDst and the calling function must allocate srcALen+srcBLen-1 words for the result.

Conceptually, when two signals a[n] and b[n] are convolved, the signal b[n] slides over a[n]. For each offset n, the overlapping portions of a[n] and b[n] are multiplied and summed together.

Note that convolution is a commutative operation:

```
a[n] * b[n] = b[n] * a[n].
```

This means that switching the A and B arguments to the convolution functions has no effect.

Fixed-Point Behavior

Convolution requires summing up a large number of intermediate products. As such, the Q7, Q15, and Q31 functions run a risk of overflow and saturation. Refer to the function specific documentation below for further details of the particular algorithm used.

Fast Versions

Fast versions are supported for Q31 and Q15. Cycles for Fast versions are less compared to Q31 and Q15 of conv and the design requires the input signals should be scaled down to avoid intermediate overflows.

Opt Versions

Opt versions are supported for Q15 and Q7. Design uses internal scratch buffer for getting good optimisation. These versions are optimised in cycles and consumes more memory(Scratch memory) compared to Q15 and Q7 versions

6.3.3 Function Docmentation

6.3.3.1 csi_conv_f32

```
void csi_conv_f32 (float32_t *pSrcA, uint32_t srcALen, float32_t *pSrcB, uint32_t␣
↪srcBLen, float32_t *pDst)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1.

Returns:

none

6.3.3.2 csi_conv_q15

```
void csi_conv_q15 (q15_t *pSrcA, uint32_t srcALen, q15_t *pSrcB, uint32_t srcBLen,␣
↪q15_t *pDst)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. Both inputs are in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. This approach provides 33 guard bits and there is no risk of overflow. The 34.30 result is then truncated to 34.15 format by discarding the low 15 bits and then saturated to 1.15 format.

Refer the function *csi_conv_opt_q15()* for a faster implementation of this function using scratch buffers.

6.3.3.3 csi_conv_q31

```
void csi_conv_q31 (q31_t *pSrcA, uint32_t srcALen, q31_t *pSrcB, uint32_t srcBLen,␣
↪q31_t *pDst)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid intermediate overflows. Scale down the inputs by log2(min(srcALen, srcBLen)) (log2 is read as log to the base 2) times to avoid overflows, as maximum of min(srcALen, srcBLen) number of additions are carried internally. The 2.62 accumulator is right shifted by 31 bits and saturated to 1.31 format to yield the final result.

6.3.3.4 csi_conv_q7

```
void csi_conv_q7 (q7_t *pSrcA, uint32_t srcALen, q7_t *pSrcB, uint32_t srcBLen, q7_t␣
↪*pDst)
```

Parameters:

*pSrcA: points to the first input sequence.

srcALen: length of the first input sequence.

*pSrcB: points to the second input sequence.

srcBLen: length of the second input sequence.

*pDst: points to the location where the output result is written. Length srcALen+srcBLen-1.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using a 32-bit internal accumulator. Both the inputs are represented in 1.7 format and multiplications yield a 2.14 result. The 2.14 intermediate results are accumulated in a 32-bit accumulator in 18.14 format. This approach provides 17 guard bits and there is no risk of overflow as long as max(srcALen, srcBLen)<131072. The 18.14 result is then truncated to 18.7 format by discarding the low 7 bits and then saturated to 1.7 format.

Refer the function *csi_conv_opt_q7()* for a faster implementation of this function.

6.3.3.5 csi_conv_fast_opt_q15

```
void csi_conv_fast_opt_q15 (q15_t *pSrcA, uint32_t srcALen, q15_t *pSrcB, uint32_t␣
↪srcBLen, q15_t *pDst, q15_t *pScratch1, q15_t *pScratch2)
```

Parameters:

*pSrcA: points to the first input sequence.

srcALen: length of the first input sequence.

*pSrcB: points to the second input sequence.

srcBLen: length of the second input sequence.

*pDst: points to the location where the output result is written. Length srcALen+srcBLen-1.

*pScratch1 points to scratch buffer of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.

*pScratch2 points to scratch buffer of size min(srcALen, srcBLen).

Returns:

none

Restrictions

If the silicon does not support unaligned memory access enable the macro UNALIGNED_SUPPORT_DISABLE In this case input, output, scratch1 and scratch2 buffers should be aligned by 32-bit

Scaling and Overflow Behavior:

This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid

intermediate overflows. Scale down the inputs by log2(min(srcALen, srcBLen)) (log2 is read as log to the base 2) times to avoid overflows, as maximum of min(srcALen, srcBLen) number of additions are carried internally. The 2.30 accumulator is right shifted by 15 bits and then saturated to 1.15 format to yield the final result.

See *csi_conv_q15()* for a slower implementation of this function which uses 64-bit accumulation to avoid wrap around distortion.

6.3.3.6 csi_conv_fast_q15

```
void csi_conv_fast_q15 (q15_t *pSrcA, uint32_t srcALen, q15_t *pSrcB, uint32_t␣
↪srcBLen, q15_t *pDst)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1.

Returns:

none

Scaling and Overflow Behavior:

This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid intermediate overflows. Scale down the inputs by log2(min(srcALen, srcBLen)) (log2 is read as log to the base 2) times to avoid overflows, as maximum of min(srcALen, srcBLen) number of additions are carried internally. The 2.30 accumulator is right shifted by 15 bits and then saturated to 1.15 format to yield the final result.

See *csi_conv_q15()* for a slower implementation of this function which uses 64-bit accumulation to avoid wrap around distortion.

6.3.3.7 csi_conv_fast_q31

```
void csi_conv_fast_q31 (q31_t *pSrcA, uint32_t srcALen, q31_t *pSrcB, uint32_t␣
↪srcBLen, q31_t *pDst)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1.

Returns:

none

Scaling and Overflow Behavior:

This function is optimized for speed at the expense of fixed-point precision and overflow protection. The result of each 1.31 x 1.31 multiplication is truncated to 2.30 format. These intermediate results are accumulated in a 32-bit register in 2.30 format. Finally, the accumulator is saturated and converted to a 1.31 result.

The fast version has the same overflow behavior as the standard version but provides less precision since it discards the low 32 bits of each multiplication result. In order to avoid overflows completely the input signals must be scaled down. Scale down the inputs by log2(min(srcALen, srcBLen)) (log2 is read as log to the base 2) times to avoid overflows, as maximum of min(srcALen, srcBLen) number of additions are carried internally.

See *csi_conv_q31()* for a slower implementation of this function which uses 64-bit accumulation to provide higher precision.

6.3.3.8 csi_conv_opt_q15

```
void csi_conv_opt_q15 (q15_t *pSrcA, uint32_t srcALen, q15_t *pSrcB, uint32_t srcBLen,
↪ q15_t *pDst, q15_t *pScratch1, q15_t *pScratch2)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1.

`*pScratch1` points to scratch buffer of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.

`*pScratch2` points to scratch buffer of size min(srcALen, srcBLen).

Returns:

none

Restrictions

If the silicon does not support unaligned memory access enable the macro UNALIGNED_SUPPORT_DISABLE In this case input, output, scratch1 and scratch2 buffers should be aligned by 32-bit

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. Both inputs are in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. This approach provides 33 guard bits and there is no risk of overflow. The 34.30 result is then truncated to 34.15 format by discarding the low 15 bits and then saturated to 1.15 format.

6.3.3.9 csi_conv_opt_q7

```
void csi_conv_opt_q7 (q7_t *pSrcA, uint32_t srcALen, q7_t *pSrcB, uint32_t srcBLen,␣
↪q7_t *pDst, q15_t *pScratch1, q15_t *pScratch2)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1.

`*pScratch1` points to scratch buffer of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.

`*pScratch2` points to scratch buffer of size min(srcALen, srcBLen).

Returns:

none

Restrictions

If the silicon does not support unaligned memory access enable the macro UNALIGNED_SUPPORT_DISABLE In this case input, output, scratch1 and scratch2 buffers should be aligned by 32-bit

Scaling and Overflow Behavior:

The function is implemented using a 32-bit internal accumulator. Both the inputs are represented in 1.7 format and multiplications yield a 2.14 result. The 2.14 intermediate results are accumulated in a 32-bit accumulator in 18.14 format. This approach provides 17 guard bits and there is no risk of overflow as long as max(srcALen, srcBLen)<131072. The 18.14 result is then truncated to 18.7 format by discarding the low 7 bits and then saturated to 1.7 format.

6.4 Partial Convolution

6.4.1 Functions

- *csi_conv_partial_f32* : Partial convolution of floating-point sequences.
- *csi_conv_partial_q31* : Partial convolution of Q31 sequences.
- *csi_conv_partial_q15* : Partial convolution of Q15 sequences.
- *csi_conv_partial_q7* : Partial convolution of Q7 sequences.
- *csi_conv_partial_fast_q31* : Partial convolution of Q31 sequences.
- *csi_conv_partial_fast_q15* : Partial convolution of Q15 sequences.
- *csi_conv_partial_fast_opt_q15* : Partial convolution of Q15 sequences.
- *csi_conv_partial_opt_q15* : Partial convolution of Q15 sequences.
- *csi_conv_partial_opt_q7* : Partial convolution of Q7 sequences.

6.4.2 Description

Description Partial Convolution is equivalent to Convolution except that a subset of the output samples is generated. Each function has two additional arguments. firstIndex specifies the starting index of the subset of output samples. numPoints is the number of output samples to compute. The function computes the output in the range [firstIndex, ⋯, firstIndex+numPoints-1]. The output array pDst contains numPoints values.

The allowable range of output indices is [0 srcALen+srcBLen-2]. If the requested subset does not fall in this range then the functions return CSKY_MATH_ARGUMENT_ERROR. Otherwise the functions return CSKY_MATH_SUCCESS.

Note

Refer *csi_conv_q31()* for details on fixed point behavior.

Fast Versions

Fast versions are supported for Q31 and Q15 of partial convolution. Cycles for Fast versions are less compared to Q31 and Q15 of partial conv and the design requires the input signals should be scaled down to avoid intermediate overflows.

Opt Versions

Opt versions are supported for Q15 and Q7. Design uses internal scratch buffer for getting good optimisation. These versions are optimised in cycles and consumes more memory(Scratch memory) compared to Q15 and Q7 versions of partial convolution

6.4.3 Function Docmentation

6.4.3.1 csi_conv_partial_f32

```
csi_status csi_conv_partial_f32 (float32_t *pSrcA, uint32_t srcALen, float32_t *pSrcB,
↪ uint32_t srcBLen, float32_t *pDst, uint32_t firstIndex, uint32_t numPoints)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written.

`firstIndex`: is the first output sample to start with.

`numPoints`: is the number of output points to be computed.

Returns:

Returns either CSKY_MATH_SUCCESS if the function completed correctly or CSKY_MATH_ARGUMENT_ERROR if the requested subset is not in the range [0 srcALen+srcBLen-2].

6.4.3.2 csi_conv_partial_q31

```
csi_status csi_conv_partial_q31 (q31_t *pSrcA, uint32_t srcALen, q31_t *pSrcB, uint32_
↪t srcBLen, q31_t *pDst, uint32_t firstIndex, uint32_t numPoints)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written.

`firstIndex`: is the first output sample to start with.

`numPoints`: is the number of output points to be computed.

Returns:

Returns either CSKY_MATH_SUCCESS if the function completed correctly or CSKY_MATH_ARGUMENT_ERROR if the requested subset is not in the range [0 srcALen+srcBLen-2].

6.4.3.3 csi_conv_partial_q15

```
csi_status csi_conv_partial_q15 (q15_t *pSrcA, uint32_t srcALen, q15_t *pSrcB, uint32_
↪t srcBLen, q15_t *pDst, uint32_t firstIndex, uint32_t numPoints)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written.

`firstIndex`: is the first output sample to start with.

`numPoints`: is the number of output points to be computed.

Returns:

Returns either CSKY_MATH_SUCCESS if the function completed correctly or CSKY_MATH_ARGUMENT_ERROR if the requested subset is not in the range [0 srcALen+srcBLen-2].

6.4.3.4 csi_conv_partial_q7

```
csi_status csi_conv_partial_q7 (q7_t *pSrcA, uint32_t srcALen, q7_t *pSrcB, uint32_t␣
↪srcBLen, q7_t *pDst, uint32_t firstIndex, uint32_t numPoints)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written.

`firstIndex`: is the first output sample to start with.

`numPoints`: is the number of output points to be computed.

Returns:

Returns either CSKY_MATH_SUCCESS if the function completed correctly or CSKY_MATH_ARGUMENT_ERROR if the requested subset is not in the range [0 srcALen+srcBLen-2].

6.4.3.5 csi_conv_partial_fast_opt_q15

```
csi_status csi_conv_partial_fast_opt_q15 (q15_t *pSrcA, uint32_t srcALen, q15_t␣
↪*pSrcB, uint32_t srcBLen, q15_t *pDst, uint32_t firstIndex, uint32_t numPoints, q15_
↪t *pScratch1, q15_t *pScratch2)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written.

`firstIndex`: is the first output sample to start with.

`numPoints`: is the number of output points to be computed.

`*pScratch1` points to scratch buffer of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.

`*pScratch2` points to scratch buffer of size min(srcALen, srcBLen).

Returns:

Returns either CSKY_MATH_SUCCESS if the function completed correctly or CSKY_MATH_ARGUMENT_ERROR if the requested subset is not in the range [0 srcALen+srcBLen-2].

See *csi_conv_partial_q15()* for a slower implementation of this function which uses a 64-bit accumulator to avoid wrap around distortion.

Restrictions:

If the silicon does not support unaligned memory access enable the macro UNALIGNED_SUPPORT_DISABLE In this case input, output, scratch1 and scratch2 buffers should be aligned by 32-bit.

6.4.3.6 csi_conv_partial_fast_q15

```
csi_status csi_conv_partial_fast_q15 (q15_t *pSrcA, uint32_t srcALen, q15_t *pSrcB,␣
↪uint32_t srcBLen, q15_t *pDst, uint32_t firstIndex, uint32_t numPoints)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written.

`firstIndex`: is the first output sample to start with.

`numPoints`: is the number of output points to be computed.

Returns:

Returns either CSKY_MATH_SUCCESS if the function completed correctly or CSKY_MATH_ARGUMENT_ERROR if the requested subset is not in the range [0 srcALen+srcBLen-2].

See *csi_conv_partial_q15()* for a slower implementation of this function which uses a 64-bit accumulator to avoid wrap around distortion.

6.4.3.7 csi_conv_partial_fast_q31

```
csi_status csi_conv_partial_fast_q31 (q31_t *pSrcA, uint32_t srcALen, q31_t *pSrcB,␣
↪uint32_t srcBLen, q31_t *pDst, uint32_t firstIndex, uint32_t numPoints)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written.

`firstIndex`: is the first output sample to start with.

`numPoints`: is the number of output points to be computed.

Returns:

Returns either CSKY_MATH_SUCCESS if the function completed correctly or CSKY_MATH_ARGUMENT_ERROR if the requested subset is not in the range [0 srcALen+srcBLen-2].

See *csi_conv_partial_q31()* for a slower implementation of this function which uses a 64-bit accumulator to avoid wrap around distortion.

6.4.3.8 csi_conv_partial_opt_q15

```
csi_status csi_conv_partial_opt_q15 (q15_t *pSrcA, uint32_t srcALen, q15_t *pSrcB,␣
↪uint32_t srcBLen, q15_t *pDst, uint32_t firstIndex, uint32_t numPoints, q15_t␣
↪*pScratch1, q15_t *pScratch2)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written.

`firstIndex`: is the first output sample to start with.

numPoints: is the number of output points to be computed.

*pScratch1 points to scratch buffer of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.

*pScratch2 points to scratch buffer of size min(srcALen, srcBLen).

Returns:

Returns either CSKY_MATH_SUCCESS if the function completed correctly or CSKY_MATH_ARGUMENT_ERROR if the requested subset is not in the range [0 srcALen+srcBLen-2].

See *csi_conv_partial_q15()* for a slower implementation of this function which uses a 64-bit accumulator to avoid wrap around distortion.

Restrictions:

If the silicon does not support unaligned memory access enable the macro UNALIGNED_SUPPORT_DISABLE In this case input, output, scratch1 and scratch2 buffers should be aligned by 32-bit.

6.4.3.9 csi_conv_partial_opt_q7

```
csi_conv_partial_opt_q7 (q7_t *pSrcA, uint32_t srcALen, q7_t *pSrcB, uint32_t srcBLen,
↪ q7_t *pDst, uint32_t firstIndex, uint32_t numPoints, q15_t *pScratch1, q15_t␣
↪*pScratch2)
```

Parameters:

*pSrcA: points to the first input sequence.

srcALen: length of the first input sequence.

*pSrcB: points to the second input sequence.

srcBLen: length of the second input sequence.

*pDst: points to the location where the output result is written.

firstIndex: is the first output sample to start with.

numPoints: is the number of output points to be computed.

*pScratch1 points to scratch buffer of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.

*pScratch2 points to scratch buffer of size min(srcALen, srcBLen).

Returns:

Returns either CSKY_MATH_SUCCESS if the function completed correctly or CSKY_MATH_ARGUMENT_ERROR if the requested subset is not in the range [0 srcALen+srcBLen-2].

Restrictions:

If the silicon does not support unaligned memory access enable the macro UNALIGNED_SUPPORT_DISABLE In this case input, output, scratch1 and scratch2 buffers should be aligned by 32-bit.

6.5 Correlation

6.5.1 Functions

- *csi_correlate_f32* : Correlation of floating-point sequences.
- *csi_correlate_q31* : Correlation of Q31 sequences.
- *csi_correlate_q15* : Correlation of Q15 sequences.
- *csi_correlate_q7* : Correlation of Q7 sequences.
- *csi_correlate_fast_q31* : Correlation of Q31 sequences.
- *csi_correlate_fast_q15* : Correlation of Q15 sequences.
- *csi_correlate_fast_opt_q15* : Correlation of Q15 sequences.
- *csi_correlate_opt_q15* : Correlation of Q15 sequences.
- *csi_correlate_opt_q7* : Correlation of Q7 sequences.

6.5.2 Description

Correlation is a mathematical operation that is similar to convolution. As with convolution, correlation uses two signals to produce a third signal. The underlying algorithms in correlation and convolution are identical except that one of the inputs is flipped in convolution. Correlation is commonly used to measure the similarity between two signals. It has applications in pattern recognition, cryptanalysis, and searching. The CSI library provides correlation functions for Q7, Q15, Q31 and floating-point data types. Fast versions of the Q15 and Q31 functions are also provided.

Algorithm

Let a[n] and b[n] be sequences of length srcALen and srcBLen samples respectively. The convolution of the two signals is denoted by

```
c[n] = a[n] * b[n]
```

In correlation, one of the signals is flipped in time

```
c[n] = a[n] * b[-n]
```

and this is mathematically defined as

$$c[n] = \sum_{k=0}^{srcALen} a[k]\, b[k-n]$$

The pSrcA points to the first input vector of length srcALen and pSrcB points to the second input vector of length srcBLen. The result c[n] is of length 2 * max(srcALen, srcBLen) - 1 and is defined over the interval n=0, 1, 2, ⋯, (2 * max(srcALen, srcBLen) - 2). The output result is written to pDst and the calling function must allocate 2 * max(srcALen, srcBLen) - 1 words for the result.

Note

The pDst should be initialized to all zeros before being used.

Fixed-Point Behavior

Correlation requires summing up a large number of intermediate products. As such, the Q7, Q15, and Q31 functions run a risk of overflow and saturation. Refer to the function specific documentation below for further details of the particular algorithm used.

Fast Versions

Fast versions are supported for Q31 and Q15. Cycles for Fast versions are less compared to Q31 and Q15 of correlate and the design requires the input signals should be scaled down to avoid intermediate overflows.

Opt Versions

Opt versions are supported for Q15 and Q7. Design uses internal scratch buffer for getting good optimisation. These versions are optimised in cycles and consumes more memory(Scratch memory) compared to Q15 and Q7 versions of correlate.

6.5.3 Function Docmentation

6.5.3.1 csi_correlate_f32

```
void csi_correlate_f32 (float32_t *pSrcA, uint32_t srcALen, float32_t *pSrcB, uint32_
↪t srcBLen, float32_t *pDst)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

Returns:

none

6.5.3.2 csi_correlate_q31

```
void csi_correlate_q31 (q31_t *pSrcA, uint32_t srcALen, q31_t *pSrcB, uint32_t␣
↪srcBLen, q31_t *pDst)
```

Parameters:

*pSrcA: points to the first input sequence.

srcALen: length of the first input sequence.

*pSrcB: points to the second input sequence.

srcBLen: length of the second input sequence.

*pDst: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid intermediate overflows. Scale down one of the inputs by 1/min(srcALen, srcBLen)to avoid overflows since a maximum of min(srcALen, srcBLen) number of additions is carried internally. The 2.62 accumulator is right shifted by 31 bits and saturated to 1.31 format to yield the final result.

6.5.3.3 csi_correlate_q15

```
void csi_correlate_q15 (q15_t *pSrcA, uint32_t srcALen, q15_t *pSrcB, uint32_t␣
↪srcBLen, q15_t *pDst)
```

Parameters:

*pSrcA: points to the first input sequence.

srcALen: length of the first input sequence.

*pSrcB: points to the second input sequence.

srcBLen: length of the second input sequence.

*pDst: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. Both inputs are in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. This approach provides 33 guard bits and there is no risk of overflow. The 34.30 result is then truncated to 34.15 format by discarding the low 15 bits and then saturated to 1.15 format.

Refer the function *csi_correlate_opt_q15()* for a faster implementation of this function using scratch buffers.

6.5.3.4 csi_correlate_q7

```
void csi_correlate_q7 (q7_t *pSrcA, uint32_t srcALen, q7_t *pSrcB, uint32_t srcBLen,␣
↪q7_t *pDst)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using a 32-bit internal accumulator. Both the inputs are represented in 1.7 format and multiplications yield a 2.14 result. The 2.14 intermediate results are accumulated in a 32-bit accumulator in 18.14 format. This approach provides 17 guard bits and there is no risk of overflow as long as max(srcALen, srcBLen)<131072. The 18.14 result is then truncated to 18.7 format by discarding the low 7 bits and saturated to 1.7 format.

Refer the function *csi_correlate_opt_q15()* for a faster implementation of this function.

6.5.3.5 csi_correlate_fast_opt_q15

```
void csi_correlate_fast_opt_q15 (q15_t *pSrcA, uint32_t srcALen, q15_t *pSrcB, uint32_
↪t srcBLen, q15_t *pDst, q15_t *pScratch)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

`*pScratch` points to scratch buffer of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.

Returns:

none

Restrictions

If the silicon does not support unaligned memory access enable the macro UNALIGNED_SUPPORT_DISABLE In this case input, output, scratch buffers should be aligned by 32-bit

Scaling and Overflow Behavior:

This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid intermediate overflows. Scale down one of the inputs by 1/min(srcALen, srcBLen) to avoid overflow since a maximum of min(srcALen, srcBLen) number of additions is carried internally. The 2.30 accumulator is right shifted by 15 bits and then saturated to 1.15 format to yield the final result.

Refer the function *csi_correlate_q15()* for a slower implementation of this function which uses a 64-bit accumulator to avoid wrap around distortion.

6.5.3.6 csi_correlate_fast_q15

```
void csi_correlate_fast_q15 (q15_t *pSrcA, uint32_t srcALen, q15_t *pSrcB, uint32_t␣
↪srcBLen, q15_t *pDst)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

Returns:

none

Scaling and Overflow Behavior:

This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid intermediate overflows. Scale down one of the inputs by 1/min(srcALen, srcBLen) to avoid overflow since a maximum of min(srcALen, srcBLen) number of additions is carried internally. The 2.30 accumulator is right shifted by 15 bits and then saturated to 1.15 format to yield the final result.

Refer the function *csi_correlate_q15()* for a slower implementation of this function which uses a 64-bit accumulator to avoid wrap around distortion.

6.5.3.7 csi_correlate_fast_q31

```
void csi_correlate_fast_q31 (q31_t *pSrcA, uint32_t srcALen, q31_t *pSrcB, uint32_t␣
↪srcBLen, q31_t *pDst)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

Returns:

none

Scaling and Overflow Behavior:

This function is optimized for speed at the expense of fixed-point precision and overflow protection. The result of each 1.31 x 1.31 multiplication is truncated to 2.30 format. These intermediate results are accumulated in a 32-bit register in 2.30 format. Finally, the accumulator is saturated and converted to a 1.31 result.

The fast version has the same overflow behavior as the standard version but provides less precision since it discards the low 32 bits of each multiplication result. In order to avoid overflows completely the input signals must be scaled down. The input signals should be scaled down to avoid intermediate overflows. Scale down one of the inputs by 1/min(srcALen, srcBLen)to avoid overflows since a maximum of min(srcALen, srcBLen) number of additions is carried internally.

Refer the function *csi_correlate_q31()* for a slower implementation of this function which uses 64-bit accumulation to provide higher precision.

6.5.3.8 csi_correlate_opt_q15

```
void csi_correlate_opt_q15 (q15_t *pSrcA, uint32_t srcALen, q15_t *pSrcB, uint32_t␣
↪srcBLen, q15_t *pDst, q15_t *pScratch)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

`*pScratch` points to scratch buffer of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.

Returns:

none

Restrictions

If the silicon does not support unaligned memory access enable the macro UNALIGNED_SUPPORT_DISABLE In this case input, output, scratch buffers should be aligned by 32-bit

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. Both inputs are in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. This approach provides 33 guard bits and there is no risk of overflow. The 34.30 result is then truncated to 34.15 format by discarding the low 15 bits and then saturated to 1.15 format.

6.5.3.9 csi_correlate_opt_q7

```
void csi_correlate_opt_q7 (q7_t *pSrcA, uint32_t srcALen, q7_t *pSrcB, uint32_t␣
↪srcBLen, q7_t *pDst, q15_t *pScratch1, q15_t *pScratch2)
```

Parameters:

`*pSrcA`: points to the first input sequence.

`srcALen`: length of the first input sequence.

`*pSrcB`: points to the second input sequence.

`srcBLen`: length of the second input sequence.

`*pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

`*pScratch` points to scratch buffer of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.

`*pScratch2` points to scratch buffer (of type q15_t) of size min(srcALen, srcBLen).

Returns:

none

Restrictions

If the silicon does not support unaligned memory access enable the macro UNALIGNED_SUPPORT_DISABLE In this case input, output, scratch1 and scratch2 buffers should be aligned by 32-bit

Scaling and Overflow Behavior:

The function is implemented using a 32-bit internal accumulator. Both the inputs are represented in 1.7 format and multiplications yield a 2.14 result. The 2.14 intermediate results are accumulated in a 32-bit accumulator in 18.14 format. This approach provides 17 guard bits and there is no risk of overflow as long as max(srcALen, srcBLen)<131072. The 18.14 result is then truncated to 18.7 format by discarding the low 7 bits and saturated to 1.7 format.

6.6 Finite Impulse Response (FIR) Filters

6.6.1 Functions

- *csi_fir_f32* : Processing function for the floating-point FIR filter.
- *csi_fir_q31* : Processing function for the Q31 FIR filter.
- *csi_fir_q15* : Processing function for the Q15 FIR filter.
- *csi_fir_q7* : Processing function for the Q7 FIR filter.
- *csi_fir_fast_q31* : Processing function for the Q31 FIR filter.
- *csi_fir_fast_q15* : Processing function for the Q15 FIR filter.
- *csi_fir_init_f32* : Initialization function for the floating-point FIR filter.
- *csi_fir_init_q31* : Initialization function for the Q31 FIR filter.
- *csi_fir_init_q15* : Initialization function for the Q15 FIR filter.
- *csi_fir_init_q7* : Initialization function for the Q7 FIR filter.

6.6.2 Description

This set of functions implements Finite Impulse Response (FIR) filters for Q7, Q15, Q31, and floating-point data types. Fast versions of Q15 and Q31 are also provided. The functions operate on blocks of input and output data and each call to the function processes blockSize samples through the filter. pSrc and pDst points to input and output arrays containing blockSize values.

Algorithm:

The FIR filter algorithm is based upon a sequence of multiply-accumulate (MAC) operations. Each filter coefficient b[n] is multiplied by a state variable which equals a previous input sample x[n].

```
y[n] = b[0] * x[n] + b[1] * x[n-1] + b[2] * x[n-2] + ...+ b[numTaps-1] * x[n-
↪numTaps+1]
```

pCoeffs points to a coefficient array of size numTaps. Coefficients are stored in time reversed order.

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

pState points to a state array of size numTaps + blockSize - 1. Samples in the state buffer are stored in the following order.

```
{x[n-numTaps+1], x[n-numTaps], x[n-numTaps-1], x[n-numTaps-2]....x[0], x[1],␣
↪..., x[blockSize-1]}
```

Note that the length of the state buffer exceeds the length of the coefficient array by blockSize-1. The increased state buffer length allows circular addressing, which is traditionally used in the FIR filters, to be avoided and yields a significant speed improvement. The state variables are updated after each block of data is processed; the coefficients are untouched.

Instance Structure

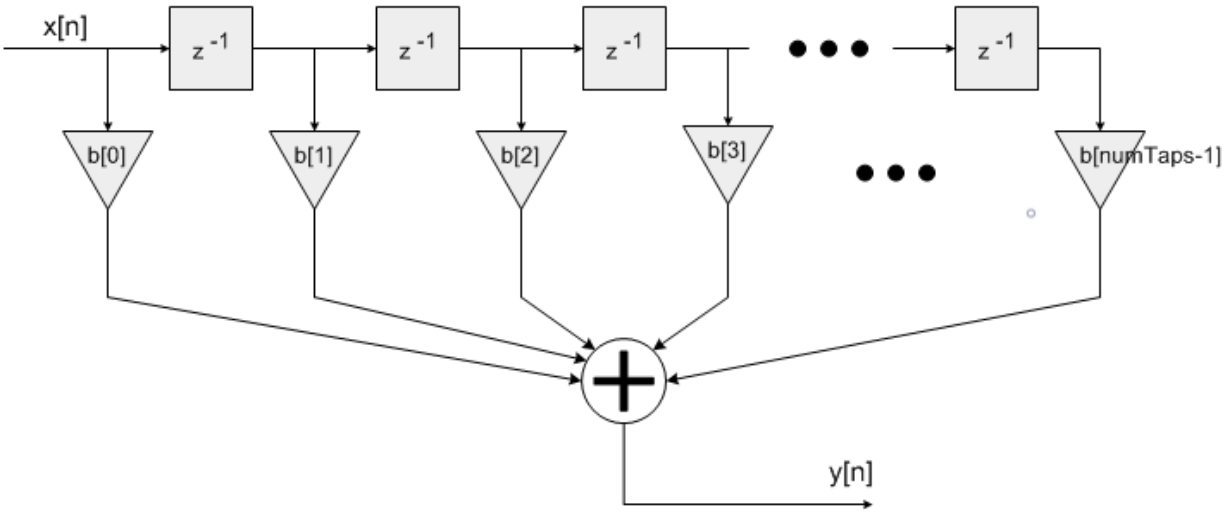


Figure6.5: Finite Impulse Response filter

The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient arrays may be shared among several instances while state variable arrays cannot be shared. There are separate instance structure declarations for each of the 4 supported data types.

Initialization Functions

There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numTaps, pCoeffs, pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros before static initialization. The code below statically initializes each of the 4 different data type filter instance structures

```
*csi_fir_instance_f32 S = {numTaps, pState, pCoeffs};
*csi_fir_instance_q31 S = {numTaps, pState, pCoeffs};
*csi_fir_instance_q15 S = {numTaps, pState, pCoeffs};
*csi_fir_instance_q7 S =  {numTaps, pState, pCoeffs};
```

where numTaps is the number of filter coefficients in the filter; pState is the address of the state buffer; pCoeffs is the address of the coefficient buffer.

Fixed-Point Behavior

Care must be taken when using the fixed-point versions of the FIR filter functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

6.6.3 Function Docmentation

6.6.3.1 csi_fir_f32

```
void csi_fir_f32 (const csi_fir_instance_f32 *S, float32_t *pSrc, float32_t *pDst,␣
↪uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR filter structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

Scaling and Overflow Behavior:

This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around and distorts the result. In order to avoid overflows completely the input signal must be scaled down by log2(numTaps) bits. The 2.30 accumulator is then truncated to 2.15 format and saturated to yield the 1.15 result.

Refer to the function *csi_fir_q15()* for a slower implementation of this function which uses 64-bit accumulation to avoid wrap around distortion. Both the slow and the fast versions use the same instance structure. Use the function *csi_fir_init_q15()* to initialize the filter structure.

6.6.3.2 csi_fir_q31

```
void csi_fir_q31 (const csi_fir_instance_q31 *S, q31_t *pSrc, q31_t *pDst, uint32_t␣
↪blockSize)
```

Parameters:

`*S`: points to an instance of the FIR filter structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clip. In order to avoid overflows completely the input signal must be scaled down by log2(numTaps) bits. After all multiply-accumulates are performed, the 2.62 accumulator is right shifted by 31 bits and saturated to 1.31 format to yield the final result.

6.6.3.3 csi_fir_q15

```
void csi_fir_q15 (const csi_fir_instance_q15 *S, q15_t *pSrc, q15_t *pDst, uint32_t␣
↪blockSize)
```

Parameters:

`*S`: points to an instance of the FIR filter structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. After all additions have been performed, the accumulator is truncated to 34.15 format by discarding low 15 bits. Lastly, the accumulator is saturated to yield a result in 1.15 format.

Refer to the function *csi_fir_fast_q15()* for a faster but less precise implementation of this function.

6.6.3.4 csi_fir_q7

```
void csi_fir_q7 (const csi_fir_instance_q7 *S, q7_t *pSrc, q7_t *pDst, uint32_t␣
↪blockSize)
```

Parameters:

`*S`: points to an instance of the FIR filter structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using a 32-bit internal accumulator. Both coefficients and state variables are represented in 1.7 format and multiplications yield a 2.14 result. The 2.14 intermediate results are accumulated in a 32-bit accumulator in 18.14 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. The accumulator is converted to 18.7 format by discarding the low 7 bits. Finally, the result is truncated to 1.7 format.

6.6.3.5 csi_fir_fast_q15

```
void csi_fir_fast_q15 (const csi_fir_instance_q15 *S, q15_t *pSrc, q15_t *pDst,␣
↪uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR filter structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

Scaling and Overflow Behavior:

This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around and distorts the result. In order to avoid overflows completely the input signal must be scaled down by log2(numTaps) bits. The 2.30 accumulator is then truncated to 2.15 format and saturated to yield the 1.15 result.

Refer to the function *csi_fir_q15()* for a slower implementation of this function which uses 64-bit accumulation to avoid wrap around distortion. Both the slow and the fast versions use the same instance structure. Use the function *csi_fir_init_q15()* to initialize the filter structure.

6.6.3.6 csi_fir_fast_q31

```
void csi_fir_fast_q31 (const csi_fir_instance_q31 *S, q31_t *pSrc, q31_t *pDst,␣
↪uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR filter structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

Scaling and Overflow Behavior:

This function is optimized for speed at the expense of fixed-point precision and overflow protection. The result of each 1.31 x 1.31 multiplication is truncated to 2.30 format. These intermediate results are added to a 2.30 accumulator. Finally, the accumulator is saturated and converted to a 1.31 result. The fast version has the same overflow behavior as the standard version and provides less precision since it discards the low 32 bits of each multiplication result. In order to avoid overflows completely the input signal must be scaled down by log2(numTaps) bits.

Refer to the function *csi_fir_q31()* for a slower implementation of this function which uses a 64-bit accumulator to provide higher precision. Both the slow and the fast versions use the same instance structure. Use the function *csi_fir_init_q31()* to initialize the filter structure.

6.6.3.7 csi_fir_init_f32

```
void csi_fir_init_f32 (csi_fir_instance_f32 *S, uint16_t numTaps, float32_t *pCoeffs,␣
↪float32_t *pState, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR filter structure.

`numTaps`: Number of filter coefficients in the filter.

`*pCoeffs`: points to the filter coefficients buffer.

`*pState`: points to the state buffer.

`blockSize`: number of samples that are processed per call.

Returns:

none

Description:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

pState points to the array of state variables. pState is of length numTaps+blockSize-1 samples, where blockSize is the number of input samples processed by each call to *csi_fir_f32()* .

6.6.3.8 csi_fir_init_q15

```
void csi_fir_init_q15 (csi_fir_instance_q15 *S, uint16_t numTaps, q15_t *pCoeffs, q15_
↪t *pState, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR filter structure.

`numTaps`: Number of filter coefficients in the filter. Must be even and greater than or equal to 4.

`*pCoeffs`: points to the filter coefficients buffer.

`*pState`: points to the state buffer.

`blockSize`: number of samples that are processed per call.

Returns:

none

Description:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

Note that numTaps must be even and greater than or equal to 4. To implement an odd length filter simply increase numTaps by 1 and set the last coefficient to zero. For example, to implement a filter with numTaps=3 and coefficients

```
{0.3, -0.8, 0.3}
```

set numTaps=4 and use the coefficients:

```
{0.3, -0.8, 0.3, 0}.
```

Similarly, to implement a two point filter

```
{0.3, -0.3}
```

set numTaps=4 and use the coefficients:

```
{0.3, -0.3, 0, 0}.
```

pState points to the array of state variables.

6.6.3.9 csi_fir_init_q31

```
void csi_fir_init_q31 (csi_fir_instance_q31 *S, uint16_t numTaps, q31_t *pCoeffs, q31_
↪t *pState, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR filter structure.

`numTaps`: Number of filter coefficients in the filter.

`*pCoeffs`: points to the filter coefficients buffer.

`*pState`: points to the state buffer.

`blockSize`: number of samples that are processed per call.

Returns:

none

Description:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

pState points to the array of state variables. pState is of length numTaps+blockSize-1 samples, where blockSize is the number of input samples processed by each call to *csi_fir_q31()* .

6.6.3.10 csi_fir_init_q7

```
void csi_fir_init_q7 (csi_fir_instance_q7 *S, uint16_t numTaps, q7_t *pCoeffs, q7_t␣
↪*pState, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR filter structure.

`numTaps`: Number of filter coefficients in the filter.

`*pCoeffs`: points to the filter coefficients buffer.

`*pState`: points to the state buffer.

`blockSize`: number of samples that are processed per call.

Returns:

none

Description:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

pState points to the array of state variables. pState is of length numTaps+blockSize-1 samples, where blockSize is the number of input samples processed by each call to *csi_fir_q7()* .

6.7 Finite Impulse Response (FIR) Decimator

6.7.1 Functions

- *csi_fir_decimate_f32* : Processing function for the floating-point FIR decimator.
- *csi_fir_decimate_q31* : Processing function for the Q31 FIR decimator.
- *csi_fir_decimate_q15* : Processing function for the Q15 FIR decimator.
- *csi_fir_decimate_fast_q31* : Processing function for the Q31 FIR decimator.
- *csi_fir_decimate_fast_q15* : Processing function for the Q15 FIR decimator.
- *csi_fir_decimate_init_f32* : Initialization function for the floating-point FIR decimator.
- *csi_fir_decimate_init_q31* : Initialization function for the Q31 FIR decimator.
- *csi_fir_decimate_init_q15* : Initialization function for the Q15 FIR decimator.

6.7.2 Description

These functions combine an FIR filter together with a decimator. They are used in multirate systems for reducing the sample rate of a signal without introducing aliasing distortion. Conceptually, the functions are equivalent to the block diagram below:

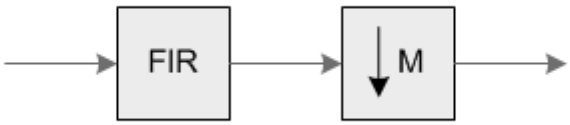


Figure6.6: Components included in the FIR Decimator functions

When decimating by a factor of M, the signal should be prefiltered by a lowpass filter with a normalized cutoff frequency of 1/M in order to prevent aliasing distortion. The user of the function is responsible for providing the filter coefficients.

The FIR decimator functions provided in the CSI DSP Library combine the FIR filter and the decimator in an efficient manner. Instead of calculating all of the FIR filter outputs and discarding M-1 out of every M, only the samples output by the decimator are computed. The functions operate on blocks of input and output data. pSrc points to an array of blockSize input values and pDst points to an array of blockSize/M output values. In order to have an integer number of output samples blockSize must always be a multiple of the decimation factor M.

The library provides separate functions for Q15, Q31 and floating-point data types.

Algorithm:

The FIR portion of the algorithm uses the standard form filter:

```
y[n] = b[0] * x[n] + b[1] * x[n-1] + b[2] * x[n-2] + ...+ b[numTaps-1] * x[n-
↪numTaps+1]
```

where, b[n] are the filter coefficients.

The pCoeffs points to a coefficient array of size numTaps. Coefficients are stored in time reversed order.

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

pState points to a state array of size numTaps + blockSize - 1. Samples in the state buffer are stored in the order:

```
{x[n-numTaps+1], x[n-numTaps], x[n-numTaps-1], x[n-numTaps-2]....x[0], x[1],␣
↪..., x[blockSize-1]}
```

The state variables are updated after each block of data is processed, the coefficients are untouched.

Instance Structure

The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient arrays may be shared among several instances while state variable array should be allocated separately. There are separate instance structure declarations for each of the 3 supported data types.

Initialization Functions

There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer.
- Checks to make sure that the size of the input is a multiple of the decimation factor. To do this manually without calling the init function, assign the follow subfields of the instance structure: numTaps, pCoeffs, M (decimation factor), pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. The code below statically initializes each of the 3 different data type filter instance structures

```
*csi_fir_decimate_instance_f32 S = {M, numTaps, pCoeffs, pState};
*csi_fir_decimate_instance_q31 S = {M, numTaps, pCoeffs, pState};
*csi_fir_decimate_instance_q15 S = {M, numTaps, pCoeffs, pState};
```

where M is the decimation factor; numTaps is the number of filter coefficients in the filter; pCoeffs is the address of the coefficient buffer; pState is the address of the state buffer. Be sure to set the values in the state buffer to zeros when doing static initialization.

Fixed-Point Behavior

Care must be taken when using the fixed-point versions of the FIR decimate filter functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

6.7.3 Function Docmentation

6.7.3.1 csi_fir_decimate_f32

```
void csi_fir_decimate_f32 (const csi_fir_decimate_instance_f32 *S, float32_t *pSrc,␣
↪float32_t *pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR decimator structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

6.7.3.2 csi_fir_decimate_q15

```
void csi_fir_decimate_q15 (const csi_fir_decimate_instance_q15 *S, q15_t *pSrc, q15_t␣
↪*pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR decimator structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. After all additions have been performed, the accumulator is truncated to 34.15 format by discarding low 15 bits. Lastly, the accumulator is saturated to yield a result in 1.15 format.

6.7.3.3 csi_fir_decimate_q31

```
void csi_fir_decimate_q31 (const csi_fir_decimate_instance_q31 *S, q31_t *pSrc, q31_t␣
↪*pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR decimator structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clip. In order to avoid overflows completely the input signal must be scaled down by log2(numTaps) bits (where log2 is read as log to the base 2). After all multiply-accumulates are performed, the 2.62 accumulator is truncated to 1.32 format and then saturated to 1.31 format.

6.7.3.4 csi_fir_decimate_fast_q15

```
void csi_fir_decimate_fast_q15 (const csi_fir_decimate_instance_q15 *S, q15_t *pSrc,␣
↪q15_t *pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR decimator structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

Restrictions

If the silicon does not support unaligned memory access enable the macro UNALIGNED_SUPPORT_DISABLE In this case input, output, state buffers should be aligned by 32-bit

Scaling and Overflow Behavior:

This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around and distorts the result. In order to avoid overflows completely the input signal must be scaled down by log2(numTaps) bits (log2 is read as log to the base 2). The 2.30 accumulator is then truncated to 2.15 format and saturated to yield the 1.15 result.

Refer to the function *csi_fir_decimate_q15()* for a slower implementation of this function which uses 64-bit accumulation to avoid wrap around distortion. Both the slow and the fast versions use the same instance structure. Use the function *csi_fir_decimate_init_q15()* to initialize the filter structure.

6.7.3.5 csi_fir_decimate_fast_q31

```
void csi_fir_decimate_fast_q31 (csi_fir_decimate_instance_q31 *S, q31_t *pSrc, q31_t␣
↪*pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR decimator structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process per call.

Returns:

none

Scaling and Overflow Behavior:

This function is optimized for speed at the expense of fixed-point precision and overflow protection. The result of each 1.31 x 1.31 multiplication is truncated to 2.30 format. These intermediate results are added to a 2.30 accumulator. Finally, the accumulator is saturated and converted to a 1.31 result. The fast version has the same overflow behavior as the standard version and provides less precision since it discards the low 32 bits of each multiplication result. In order to avoid overflows completely the input signal must be scaled down by log2(numTaps) bits (where log2 is read as log to the base 2).

Refer to the function *csi_fir_decimate_q31()* for a slower implementation of this function which uses a 64-bit accumulator to provide higher precision. Both the slow and the fast versions use the same instance structure. Use the function *csi_fir_decimate_init_q31()* to initialize the filter structure.

6.7.3.6 csi_fir_decimate_init_f32

```
csi_status csi_fir_decimate_init_f32 (csi_fir_decimate_instance_f32 *S, uint16_t␣
↪numTaps, uint8_t M, float32_t *pCoeffs, float32_t *pState, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR decimator structure.

`numTaps`: number of coefficients in the filter.

`M`: decimation factor.

`*pCoeffs`: points to the filter coefficients.

`*pState`: points to the state buffer.

`blockSize`: number of samples to process per call.

Returns:

The function returns CSKY_MATH_SUCCESS if initialization was successful or CSKY_MATH_LENGTH_ERROR if blockSize is not a multiple of M.

Description:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

pState points to the array of state variables. pState is of length numTaps+blockSize-1 words where blockSize is the number of input samples passed to *csi_fir_decimate_f32()* . M is the decimation factor.

6.7.3.7 csi_fir_decimate_init_q15

```
csi_status csi_fir_decimate_init_q15 (csi_fir_decimate_instance_q15 *S, uint16_t␣
↪numTaps, uint8_t M, q15_t *pCoeffs, q15_t *pState, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR decimator structure.

`numTaps`: number of coefficients in the filter.

`M`: decimation factor.

`*pCoeffs`: points to the filter coefficients.

`*pState`: points to the state buffer.

`blockSize`: number of samples to process per call.

Returns:

The function returns CSKY_MATH_SUCCESS if initialization was successful or CSKY_MATH_LENGTH_ERROR if blockSize is not a multiple of M.

Description:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

pState points to the array of state variables. pState is of length numTaps+blockSize-1 words where blockSize is the number of input samples to the call *csi_fir_decimate_q15()* . M is the decimation factor.

6.7.3.8 csi_fir_decimate_init_q31

```
csi_status csi_fir_decimate_init_q31 (csi_fir_decimate_instance_q31 *S, uint16_t␣
↪numTaps, uint8_t M, q31_t *pCoeffs, q31_t *pState, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR decimator structure.

`numTaps`: number of coefficients in the filter.

`M`: decimation factor.

`*pCoeffs`: points to the filter coefficients.

`*pState`: points to the state buffer.

`blockSize`: number of samples to process per call.

Returns:

The function returns CSKY_MATH_SUCCESS if initialization was successful or CSKY_MATH_LENGTH_ERROR if blockSize is not a multiple of M.

Description:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

pState points to the array of state variables. pState is of length numTaps+blockSize-1 words where blockSize is the number of input samples to the call *csi_fir_decimate_q31()* . M is the decimation factor.

6.8 Finite Impulse Response (FIR) Lattice Filters

6.8.1 Functions

- *csi_fir_lattice_f32* : Processing function for the floating-point FIR lattice filter.
- *csi_fir_lattice_q31* : Processing function for the Q31 FIR lattice filter.
- *csi_fir_lattice_q15* : Processing function for the Q15 FIR lattice filter.
- *csi_fir_lattice_init_f32* : Initialization function for the floating-point FIR lattice filter.
- *csi_fir_lattice_init_q31* : Initialization function for the Q31 FIR lattice filter.
- *csi_fir_lattice_init_q15* : Initialization function for the Q15 FIR lattice filter.

6.8.2 Description

This set of functions implements Finite Impulse Response (FIR) lattice filters for Q15, Q31 and floating-point data types. Lattice filters are used in a variety of adaptive filter applications. The filter structure is feedforward and the net impulse response is finite length. The functions operate on blocks of input and output data and each call to the function processes blockSize samples through the filter. pSrc and pDst point to input and output arrays containing blockSize values.

Algorithm:

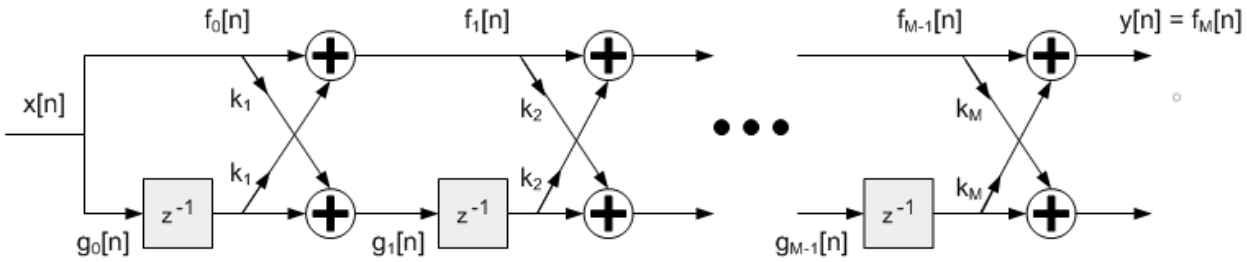


Figure6.7: Finite Impulse Response Lattice filter

The following difference equation is implemented:

```
f0[n] = g0[n] = x[n]
fm[n] = fm-1[n] + km * gm-1[n-1] for m = 1, 2, ...M
gm[n] = km * fm-1[n] + gm-1[n-1] for m = 1, 2, ...M
y[n] = fM[n]
```

pCoeffs points to tha array of reflection coefficients of size numStages. Reflection Coefficients are stored in the following order.

```
{k1, k2, ..., kM}
```

where M is number of stages

pState points to a state array of size numStages. The state variables (g values) hold previous inputs and are stored in the following order.

```
{g0[n], g1[n], g2[n] ...gM-1[n]}
```

The state variables are updated after each block of data is processed; the coefficients are untouched.

Instance Structure

The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient arrays may be shared among several instances while state variable arrays cannot be shared. There are separate instance structure declarations for each of the 3 supported data types.

Initialization Functions

There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numStages, pCoeffs, pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros and then manually initialize the instance structure as follows:

```
*csi_fir_lattice_instance_f32 S = {numStages, pState, pCoeffs};
*csi_fir_lattice_instance_q31 S = {numStages, pState, pCoeffs};
*csi_fir_lattice_instance_q15 S = {numStages, pState, pCoeffs};
```

where numStages is the number of stages in the filter; pState is the address of the state buffer; pCoeffs is the address of the coefficient buffer.

Fixed-Point Behavior

Care must be taken when using the fixed-point versions of the FIR Lattice filter functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

6.8.3 Function Docmentation

6.8.3.1 csi_fir_lattice_f32

```
void csi_fir_lattice_f32 (const csi_fir_lattice_instance_f32 *S, float32_t *pSrc,␣
↪float32_t *pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR lattice structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process.

Returns:

none

6.8.3.2 csi_fir_lattice_q31

```
void csi_fir_lattice_q31 (const csi_fir_lattice_instance_q31 *S, q31_t *pSrc, q31_t␣
↪*pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR lattice structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process.

Returns:

none

Scaling and Overflow Behavior: In order to avoid overflows the input signal must be scaled down by 2*log2(numStages) bits.

6.8.3.3 csi_fir_lattice_q15

```
void csi_fir_lattice_q15 (const csi_fir_lattice_instance_q15 *S, q15_t *pSrc, q15_t␣
↪*pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR lattice structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process.

Returns:

none

6.8.3.4 csi_fir_lattice_init_f32

```
void csi_fir_lattice_init_f32 (csi_fir_lattice_instance_f32 *S, uint16_t numStages,␣
↪float32_t *pCoeffs, float32_t *pState)
```

Parameters:

`*S`: points to an instance of the FIR lattice structure.

`numStages`: number of filter stages.

`*pCoeffs`: points to the coefficient buffer. The array is of length numStages.

`*pState`: points to the state buffer. The array is of length numStages.

Returns:

none

6.8.3.5 csi_fir_lattice_init_q31

```
void csi_fir_lattice_init_q31 (csi_fir_lattice_instance_q31 *S, uint16_t numStages,␣
↪q31_t *pCoeffs, q31_t *pState)
```

Parameters:

`*S`: points to an instance of the FIR lattice structure.

`numStages`: number of filter stages.

`*pCoeffs`: points to the coefficient buffer. The array is of length numStages.

`*pState`: points to the state buffer. The array is of length numStages.

Returns:

none

6.8.3.6 csi_fir_lattice_init_q15

```
void csi_fir_lattice_init_q15 (csi_fir_lattice_instance_q15 *S, uint16_t numStages,␣
↪q15_t *pCoeffs, q15_t *pState)
```

Parameters:

`*S`: points to an instance of the FIR lattice structure.

`numStages`: number of filter stages.

`*pCoeffs`: points to the coefficient buffer. The array is of length numStages.

`*pState`: points to the state buffer. The array is of length numStages.

Returns:

none

6.9 Finite Impulse Response (FIR) Sparse Filters

6.9.1 Functions

- *csi_fir_sparse_f32* : Processing function for the floating-point FIR Sparse filter.
- *csi_fir_sparse_q31* : Processing function for the Q31 FIR Sparse filter.
- *csi_fir_sparse_q15* : Processing function for the Q15 FIR Sparse filter.
- *csi_fir_sparse_q7* : Processing function for the Q15 FIR Sparse filter.
- *csi_fir_sparse_init_f32* : Initialization function for the floating-point FIR Sparse filter.
- *csi_fir_sparse_init_q31* : Initialization function for the Q31 FIR Sparse filter.
- *csi_fir_sparse_init_q15* : Initialization function for the Q15 FIR Sparse filter.
- *csi_fir_sparse_init_q7* : Initialization function for the Q7 FIR Sparse filter.

6.9.2 Description

This group of functions implements sparse FIR filters. Sparse FIR filters are equivalent to standard FIR filters except that most of the coefficients are equal to zero. Sparse filters are used for simulating reflections in communications and audio applications.

There are separate functions for Q7, Q15, Q31, and floating-point data types. The functions operate on blocks of input and output data and each call to the function processes blockSize samples through the filter. pSrc and pDst points to input and output arrays respectively containing blockSize values.

Algorithm:

The sparse filter instant structure contains an array of tap indices pTapDelay which specifies the locations of the non-zero coefficients. This is in addition to the coefficient array b. The implementation essentially skips the multiplications by zero and leads to an efficient realization.

```
y[n] = b[0] * x[n-pTapDelay[0]] + b[1] * x[n-pTapDelay[1]] + b[2] * x[n-
↪pTapDelay[2]] + ...+ b[numTaps-1] * x[n-pTapDelay[numTaps-1]]
```

pCoeffs points to a coefficient array of size numTaps; pTapDelay points to an array of nonzero indices and is also of size numTaps; pState points to a state array of size maxDelay + blockSize, where maxDelay is the largest offset value that is ever used in the pTapDelay array. Some of the processing functions also require temporary working buffers.

Instance Structure

The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient and offset arrays may be shared among several instances while state variable arrays cannot be shared. There are separate instance structure declarations for each of the 4 supported data types.

Initialization Functions

There is also an associated initialization function for each data type. The initialization function performs the following operations:

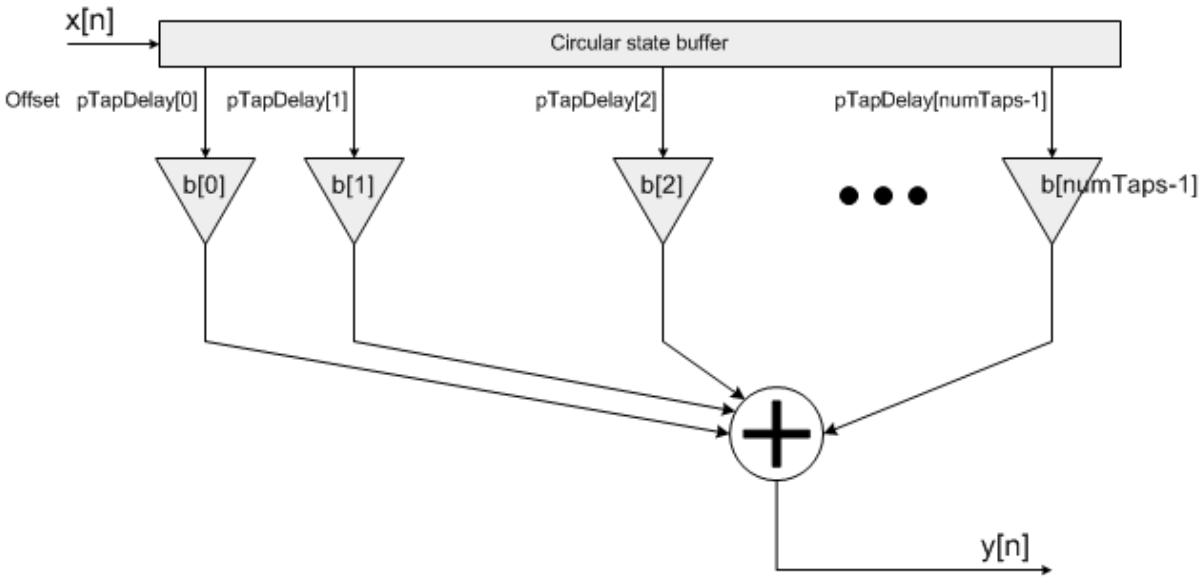


Figure6.8: Sparse FIR filter. b[n] represents the filter coefficients

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numTaps, pCoeffs, pTapDelay, maxDelay, stateIndex, pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros before static initialization. The code below statically initializes each of the 4 different data type filter instance structures

```
*csi_fir_sparse_instance_f32 S = {numTaps, 0, pState, pCoeffs, maxDelay,␣
↪pTapDelay};
*csi_fir_sparse_instance_q31 S = {numTaps, 0, pState, pCoeffs, maxDelay,␣
↪pTapDelay};
*csi_fir_sparse_instance_q15 S = {numTaps, 0, pState, pCoeffs, maxDelay,␣
↪pTapDelay};
*csi_fir_sparse_instance_q7 S =  {numTaps, 0, pState, pCoeffs, maxDelay,␣
↪pTapDelay};
```

Fixed-Point Behavior

Care must be taken when using the fixed-point versions of the sparse FIR filter functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

6.9.3 Function Docmentation

6.9.3.1 csi_fir_sparse_f32

```
void csi_fir_sparse_f32 (csi_fir_sparse_instance_f32 *S, float32_t *pSrc, float32_t␣
↪*pDst, float32_t *pScratchIn, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR sparse structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`*pScratchIn`: points to a temporary buffer of size blockSize.

`blockSize`: number of samples to process.

Returns:

none

6.9.3.2 csi_fir_sparse_q31

```
void csi_fir_sparse_q31 (csi_fir_sparse_instance_q31 *S, q31_t *pSrc, q31_t *pDst,␣
↪q31_t *pScratchIn, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR sparse structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`*pScratchIn`: points to a temporary buffer of size blockSize.

`blockSize`: number of samples to process.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using an internal 32-bit accumulator. The 1.31 x 1.31 multiplications are truncated to 2.30 format. This leads to loss of precision on the intermediate multiplications and provides only a single guard bit. If the accumulator result overflows, it wraps around rather than saturate. In order to avoid overflows the input signal or coefficients must be scaled down by log2(numTaps) bits.

6.9.3.3 csi_fir_sparse_q15

```
void csi_fir_sparse_q15 (csi_fir_sparse_instance_q15 *S, q15_t *pSrc, q15_t *pDst,␣
↪q15_t *pScratchIn, q31_t *pScratchOut, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR sparse structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`*pScratchIn`: points to a temporary buffer of size blockSize.

`*pScratchOut`: points to a temporary buffer of size blockSize.

`blockSize`: number of samples to process.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using an internal 32-bit accumulator. The 1.15 x 1.15 multiplications yield a 2.30 result and these are added to a 2.30 accumulator. Thus the full precision of the multiplications is maintained but there is only a single guard bit in the accumulator. If the accumulator result overflows it will wrap around rather than saturate. After all multiply-accumulates are performed, the 2.30 accumulator is truncated to 2.15 format and then saturated to 1.15 format. In order to avoid overflows the input signal or coefficients must be scaled down by log2(numTaps) bits.

6.9.3.4 csi_fir_sparse_q7

```
void csi_fir_sparse_q7 (csi_fir_sparse_instance_q7 *S, q7_t *pSrc, q7_t *pDst, q7_t␣
↪*pScratchIn, q31_t *pScratchOut, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR sparse structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`*pScratchIn`: points to a temporary buffer of size blockSize.

`*pScratchOut`: points to a temporary buffer of size blockSize.

`blockSize`: number of samples to process.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using a 32-bit internal accumulator. Both coefficients and state variables are represented in 1.7 format and multiplications yield a 2.14 result. The 2.14 intermediate results are accumulated in a 32-bit accumulator in 18.14 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. The accumulator is then converted to 18.7 format by discarding the low 7 bits. Finally, the result is truncated to 1.7 format.

6.9.3.5 csi_fir_sparse_init_f32

```
void csi_fir_sparse_init_f32 (csi_fir_sparse_instance_f32 *S, uint16_t numTaps,␣
↪float32_t *pCoeffs, float32_t *pState, int32_t *pTapDelay, uint16_t maxDelay,␣
↪uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR sparse structure.

`numTaps`: number of nonzero coefficients in the filter.

`*pCoeffs`: points to the array of filter coefficients.

`*pState`: points to the state buffer.

`*pTapDelay`: points to the array of offset times.

`maxDelay`: maximum offset time supported.

`blockSize`: number of samples to process.

Returns:

none

Description:

pCoeffs holds the filter coefficients and has length numTaps. pState holds the filter's state variables and must be of length maxDelay + blockSize, where maxDelay is the maximum number of delay line values. blockSize is the number of samples processed by the *csi_fir_sparse_f32()* function.

6.9.3.6 csi_fir_sparse_init_q31

```
void csi_fir_sparse_init_q31 (csi_fir_sparse_instance_q31 *S, uint16_t numTaps, q31_t␣
↪*pCoeffs, q31_t *pState, int32_t *pTapDelay, uint16_t maxDelay, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR sparse structure.

`numTaps`: number of nonzero coefficients in the filter.

`*pCoeffs`: points to the array of filter coefficients.

`*pState`: points to the state buffer.

`*pTapDelay`: points to the array of offset times.

`maxDelay`: maximum offset time supported.

`blockSize`: number of samples to process.

Returns:

none

Description:

pCoeffs holds the filter coefficients and has length numTaps. pState holds the filter's state variables and must be of length maxDelay + blockSize, where maxDelay is the maximum number of delay line values. blockSize is the number of words processed by *csi_fir_sparse_q31()* function.

6.9.3.7 csi_fir_sparse_init_q15

```
void csi_fir_sparse_init_q15 (csi_fir_sparse_instance_q15 *S, uint16_t numTaps, q15_t␣
↪*pCoeffs, q15_t *pState, int32_t *pTapDelay, uint16_t maxDelay, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR sparse structure.

`numTaps`: number of nonzero coefficients in the filter.

`*pCoeffs`: points to the array of filter coefficients.

`*pState`: points to the state buffer.

`*pTapDelay`: points to the array of offset times.

`maxDelay`: maximum offset time supported.

`blockSize`: number of samples to process.

Returns:

none

Description:

pCoeffs holds the filter coefficients and has length numTaps. pState holds the filter's state variables and must be of length maxDelay + blockSize, where maxDelay is the maximum number of delay line values. blockSize is the number of words processed by *csi_fir_sparse_q15()* function.

6.9.3.8 csi_fir_sparse_init_q7

```
void csi_fir_sparse_init_q7 (csi_fir_sparse_instance_q7 *S, uint16_t numTaps, q7_t␣
↪*pCoeffs, q7_t *pState, int32_t *pTapDelay, uint16_t maxDelay, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR sparse structure.

`numTaps`: number of nonzero coefficients in the filter.

`*pCoeffs`: points to the array of filter coefficients.

`*pState`: points to the state buffer.

`*pTapDelay`: points to the array of offset times.

`maxDelay`: maximum offset time supported.

`blockSize`: number of samples to process.

Returns:

none

Description:

pCoeffs holds the filter coefficients and has length numTaps. pState holds the filter's state variables and must be of length maxDelay + blockSize, where maxDelay is the maximum number of delay line values. blockSize is the number of words processed by *csi_fir_sparse_q7()* function.

6.10 Finite Impulse Response (FIR) interpolator Filters

6.10.1 Functions

- *csi_fir_interpolate_f32* : Processing function for the floating-point FIR interpolator filter.
- *csi_fir_interpolate_q31* : Processing function for the Q31 FIR interpolator filter.
- *csi_fir_interpolate_q15* : Processing function for the Q15 FIR interpolator filter.
- *csi_fir_interpolate_init_f32* : Initialization function for the floating-point FIR interpolator filter.
- *csi_fir_interpolate_init_q31* : Initialization function for the Q31 FIR interpolator filter.
- *csi_fir_interpolate_init_q15* : Initialization function for the Q15 FIR interpolator filter.

6.10.2 Description

These functions combine an upsampler (zero stuffer) and an FIR filter. They are used in multirate systems for increasing the sample rate of a signal without introducing high frequency images. Conceptually, the functions are equivalent to the block diagram below:

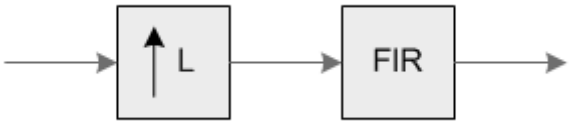


Figure6.9: Components included in the FIR Interpolator functions

After upsampling by a factor of L, the signal should be filtered by a lowpass filter with a normalized cutoff frequency of 1/L in order to eliminate high frequency copies of the spectrum. The user of the function is responsible for providing the filter coefficients.

The FIR interpolator functions provided in the CSI DSP Library combine the upsampler and FIR filter in an efficient manner. The upsampler inserts L-1 zeros between each sample. Instead of multiplying by these zero values, the FIR filter is designed to skip them. This leads to an efficient implementation without any wasted effort. The functions operate on blocks of input and output data. pSrc points to an array of blockSize input values and pDst points to an array of blockSize*L output values.

The library provides separate functions for Q15, Q31, and floating-point data types.

Algorithm:

The functions use a polyphase filter structure:

```
y[n] = b[0] * x[n] + b[L] * x[n-1] + ... + b[L*(phaseLength-1)] * x[n-
↪phaseLength+1]
y[n+1] = b[1] * x[n] + b[L+1] * x[n-1] + ... + b[L*(phaseLength-1)+1] * x[n-
↪phaseLength+1]
```

(continues on next page)

(continued from previous page)

```
...
y[n+(L-1)] = b[L-1] * x[n] + b[2*L-1] * x[n-1] + ....+ b[L*(phaseLength-
↪1)+(L-1)] * x[n-phaseLength+1]
```

This approach is more efficient than straightforward upsample-then-filter algorithms. With this method the computation is reduced by a factor of 1/L when compared to using a standard FIR filter.

pCoeffs points to a coefficient array of size numTaps. numTaps must be a multiple of the interpolation factor L and this is checked by the initialization functions. Internally, the function divides the FIR filter's impulse response into shorter filters of length phaseLength=numTaps/L. Coefficients are stored in time reversed order.

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

pState points to a state array of size blockSize + phaseLength - 1. Samples in the state buffer are stored in the order:

```
{x[n-phaseLength+1], x[n-phaseLength], x[n-phaseLength-1], x[n-phaseLength-
↪2]....x[0], x[1], ..., x[blockSize-1]}
```

The state variables are updated after each block of data is processed, the coefficients are untouched.

Instance Structure

The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient arrays may be shared among several instances while state variable array should be allocated separately. There are separate instance structure declarations for each of the 3 supported data types.

Initialization Functions

There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer.
- Checks to make sure that the length of the filter is a multiple of the interpolation factor. To do this manually without calling the init function, assign the follow subfields of the instance structure: L (interpolation factor), pCoeffs, phaseLength (numTaps / L), pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. The code below statically initializes each of the 3 different data type filter instance structures

```
csi_fir_interpolate_instance_f32 S = {L, phaseLength, pCoeffs, pState};
csi_fir_interpolate_instance_q31 S = {L, phaseLength, pCoeffs, pState};
csi_fir_interpolate_instance_q15 S = {L, phaseLength, pCoeffs, pState};
```

where L is the interpolation factor; phaseLength=numTaps/L is the length of each of the shorter FIR filters used internally, pCoeffs is the address of the coefficient buffer; pState is the address of the state buffer. Be sure to set the values in the state buffer to zeros when doing static initialization.

Fixed-Point Behavior

Care must be taken when using the fixed-point versions of the FIR interpolate filter functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

6.10.3 Function Docmentation

6.10.3.1 csi_fir_interpolate_f32

```
void csi_fir_interpolate_f32 (const csi_fir_interpolate_instance_f32 *S, float32_t␣
↪*pSrc, float32_t *pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR interpolator structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process.

Returns:

none

6.10.3.2 csi_fir_interpolate_q31

```
void csi_fir_interpolate_q31 (const csi_fir_interpolate_instance_q31 *S, q31_t *pSrc,␣
↪q31_t *pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR interpolator structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clip. In order to avoid overflows completely the input signal must be scaled down by 1/(numTaps/L). since numTaps/L additions occur per output sample. After all multiply-accumulates are performed, the 2.62 accumulator is truncated to 1.32 format and then saturated to 1.31 format.

6.10.3.3 csi_fir_interpolate_q15

```
void csi_fir_interpolate_q15 (const csi_fir_interpolate_instance_q15 *S, q15_t *pSrc,␣
↪q15_t *pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR interpolator structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. After all additions have been performed, the accumulator is truncated to 34.15 format by discarding low 15 bits. Lastly, the accumulator is saturated to yield a result in 1.15 format.

6.10.3.4 csi_fir_interpolate_init_f32

```
void csi_fir_interpolate_init_f32 (csi_fir_interpolate_instance_f32 *S, uint8_t L,␣
↪uint16_t numTaps, float32_t *pCoeffs, float32_t *pState, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR interpolator structure.

`L`: upsample factor.

`numTaps`: number of filter coefficients in the filter.

`*pCoeffs`: points to the filter coefficient buffer.

`*pState`: points to the state buffer.

`blockSize`: number of samples to process.

Returns:

none

Description:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[numTaps-2], ..., b[1], b[0]}
```

The length of the filter numTaps must be a multiple of the interpolation factor L.

pState points to the array of state variables. pState is of length (numTaps/L)+blockSize-1 words where blockSize is the number of input samples processed by each call to *csi_fir_interpolate_f32()* .

6.10.3.5 csi_fir_interpolate_init_q31

```
void csi_fir_interpolate_init_q31 (csi_fir_interpolate_instance_q31 *S, uint8_t L,␣
↪uint16_t numTaps, q31_t *pCoeffs, q31_t *pState, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR interpolator structure.

`L`: upsample factor.

`numTaps`: number of filter coefficients in the filter.

`*pCoeffs`: points to the filter coefficient buffer.

`*pState`: points to the state buffer.

`blockSize`: number of samples to process.

Returns:

none

Description:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[numTaps-2], ..., b[1], b[0]}
```

The length of the filter numTaps must be a multiple of the interpolation factor L.

pState points to the array of state variables. pState is of length (numTaps/L)+blockSize-1 words where blockSize is the number of input samples processed by each call to *csi_fir_interpolate_q31()* .

6.10.3.6 csi_fir_interpolate_init_q15

```
void csi_fir_interpolate_init_q15 (csi_fir_interpolate_instance_q15 *S, uint8_t L,␣
↪uint16_t numTaps, q15_t *pCoeffs, q15_t *pState, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the FIR interpolator structure.

`L`: upsample factor.

`numTaps`: number of filter coefficients in the filter.

`*pCoeffs`: points to the filter coefficient buffer.

`*pState`: points to the state buffer.

`blockSize`: number of samples to process.

Returns:

none

Description:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[numTaps-2], ..., b[1], b[0]}
```

The length of the filter numTaps must be a multiple of the interpolation factor L.

pState points to the array of state variables. pState is of length (numTaps/L)+blockSize-1 words where blockSize is the number of input samples processed by each call to *csi_fir_interpolate_q15()* .

6.11 Infinite Impulse Response (IIR) Lattice Filters

6.11.1 Functions

- *csi_iir_lattice_f32* : Processing function for the floating-point IIR lattice filter.
- *csi_iir_lattice_q31* : Processing function for the Q31 IIR lattice filter.
- *csi_iir_lattice_q15* : Processing function for the Q15 IIR lattice filter.
- *csi_iir_lattice_init_f32* : Initialization function for the floating-point IIR lattice filter.
- *csi_iir_lattice_init_q31* : Initialization function for the Q31 IIR lattice filter.
- *csi_iir_lattice_init_q15* : Initialization function for the Q15 IIR lattice filter.

6.11.2 Description

This set of functions implements lattice filters for Q15, Q31 and floating-point data types. Lattice filters are used in a variety of adaptive filter applications. The filter structure has feedforward and feedback components and the net impulse response is infinite length. The functions operate on blocks of input and output data and each call to the function processes blockSize samples through the filter. pSrc and pDst point to input and output arrays containing blockSize values.

Algorithm:

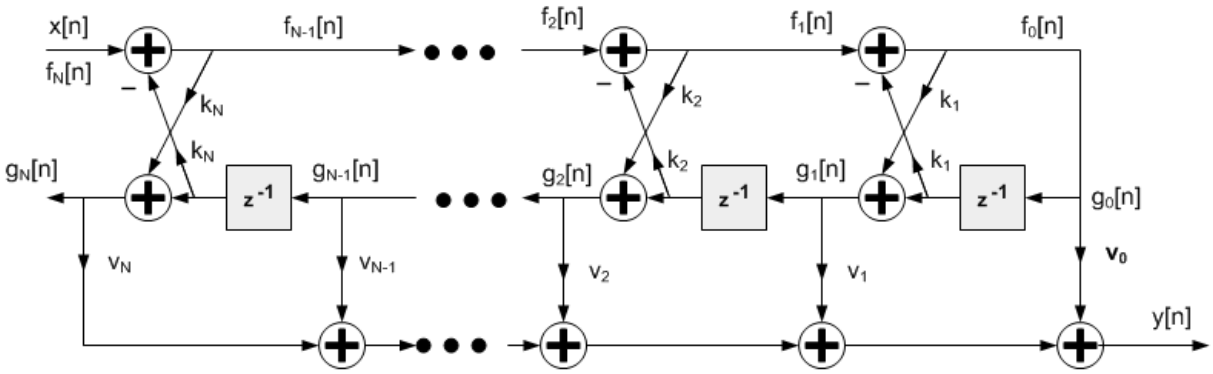


Figure6.10: Infinite Impulse Response Lattice filter

```
fN(n)    = x(n)
fm-1(n)  = fm(n) - km * gm-1(n-1) for m = N, N-1, ...1
gm(n)    = km * fm-1(n) + gm-1(n-1) for m = N, N-1, ...1
y(n)     = vN * gN(n) + vN-1 * gN-1(n) + ...+ v0 * g0(n)
```

pkCoeffs points to array of reflection coefficients of size numStages. Reflection coefficients are stored in time-reversed order.

```
{kN, kN-1, ....k1}
```

pvCoeffs points to the array of ladder coefficients of size (numStages+1). Ladder coefficients are stored in time-reversed order.

```
{vN, vN-1, ...v0}
```

pState points to a state array of size numStages + blockSize. The state variables shown in the figure above (the g values) are stored in the pState array. The state variables are updated after each block of data is processed; the coefficients are untouched.

Instance Structure

The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient arrays may be shared among several instances while state variable arrays cannot be shared. There are separate instance structure declarations for each of the 3 supported data types.

Initialization Functions

There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numStages, pkCoeffs, pvCoeffs, pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros and then manually initialize the instance structure as follows:

```
*csi_iir_lattice_instance_f32 S = {numStages, pState, pkCoeffs, pvCoeffs};
*csi_iir_lattice_instance_q31 S = {numStages, pState, pkCoeffs, pvCoeffs};
*csi_iir_lattice_instance_q15 S = {numStages, pState, pkCoeffs, pvCoeffs};
```

where numStages is the number of stages in the filter; pState points to the state buffer array; pkCoeffs points to array of the reflection coefficients; pvCoeffs points to the array of ladder coefficients.

Fixed-Point Behavior

Care must be taken when using the fixed-point versions of the IIR lattice filter functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

6.11.3 Function Docmentation

6.11.3.1 csi_iir_lattice_f32

```
void csi_iir_lattice_f32 (const csi_iir_lattice_instance_f32 *S, float32_t *pSrc,␣
↪float32_t *pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the IIR lattice structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process.

Returns:

none

6.11.3.2 csi_iir_lattice_q31

```
void csi_iir_lattice_q31 (const csi_iir_lattice_instance_q31 *S, q31_t *pSrc, q31_t␣
↪*pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the IIR lattice structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clip. In order to avoid overflows completely the input signal must be scaled down by 2*log2(numStages) bits. After all multiply-accumulates are performed, the 2.62 accumulator is saturated to 1.32 format and then truncated to 1.31 format.

6.11.3.3 csi_iir_lattice_q15

```
void csi_iir_lattice_q15 (const csi_iir_lattice_instance_q15 *S, q15_t *pSrc, q15_t␣
↪*pDst, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the IIR lattice structure.

`*pSrc`: points to the block of input data.

`*pDst`: points to the block of output data.

`blockSize`: number of samples to process.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. After all additions have been performed, the accumulator is truncated to 34.15 format by discarding low 15 bits. Lastly, the accumulator is saturated to yield a result in 1.15 format.

6.11.3.4 csi_iir_lattice_init_f32

```
void csi_iir_lattice_init_f32 (csi_iir_lattice_instance_f32 *S, uint16_t numStages,␣
↪float32_t *pkCoeffs, float32_t *pvCoeffs, float32_t *pState, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the IIR lattice structure.

`numStages`: number of stages in the filter.

`*pkCoeffs`: points to the reflection coefficient buffer. The array is of length numStages.

`*pvCoeffs`: points to the ladder coefficient buffer. The array is of length numStages+1.

`*pState`: points to the state buffer. The array is of length numStages+blockSize.

`blockSize`: number of samples to process.

Returns:

none

6.11.3.5 csi_iir_lattice_init_q31

```
void csi_iir_lattice_init_q31 (csi_iir_lattice_instance_q31 *S, uint16_t numStages,␣
↪q31_t *pkCoeffs, q31_t *pvCoeffs, q31_t *pState, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the IIR lattice structure.

`numStages`: number of stages in the filter.

`*pkCoeffs`: points to the reflection coefficient buffer. The array is of length numStages.

`*pvCoeffs`: points to the ladder coefficient buffer. The array is of length numStages+1.

`*pState`: points to the state buffer. The array is of length numStages+blockSize.

`blockSize`: number of samples to process.

Returns:

none

6.11.3.6 csi_iir_lattice_init_q15

```
void csi_iir_lattice_init_q15 (csi_iir_lattice_instance_q15 *S, uint16_t numStages,␣
↪q15_t *pkCoeffs, q15_t *pvCoeffs, q15_t *pState, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the IIR lattice structure.

`numStages`: number of stages in the filter.

`*pkCoeffs`: points to the reflection coefficient buffer. The array is of length numStages.

`*pvCoeffs`: points to the ladder coefficient buffer. The array is of length numStages+1.

`*pState`: points to the state buffer. The array is of length numStages+blockSize.

`blockSize`: number of samples to process.

Returns:

none

6.12 Least Mean Square (LMS) Filters

6.12.1 Functions

- *csi_lms_f32* : Processing function for the floating-point LMS filter.
- *csi_lms_q31* : Processing function for the Q31 LMS filter.
- *csi_lms_q15* : Processing function for the Q15 LMS filter.
- *csi_lms_init_f32* : Initialization function for the floating-point LMS filter.
- *csi_lms_init_q31* : Initialization function for the Q31 LMS filter.
- *csi_lms_init_q15* : Initialization function for the Q15 LMS filter.

6.12.2 Description

LMS filters are a class of adaptive filters that are able to "learn" an unknown transfer functions. LMS filters use a gradient descent method in which the filter coefficients are updated based on the instantaneous error signal. Adaptive filters are often used in communication systems, equalizers, and noise removal. The CSI DSP Library contains LMS filter functions that operate on Q15, Q31, and floating-point data types. The library also contains normalized LMS filters in which the filter coefficient adaptation is indepedent of the level of the input signal.

An LMS filter consists of two components as shown below. The first component is a standard transversal or FIR filter. The second component is a coefficient update mechanism. The LMS filter has two input signals. The "input" feeds the FIR filter while the "reference input" corresponds to the desired output of the FIR filter. That is, the FIR filter coefficients are updated so that the output of the FIR filter matches the reference input. The filter coefficient update mechanism is based on the difference between the FIR filter output and the reference input. This "error signal" tends towards zero as the filter adapts. The LMS processing functions accept the input and reference input signals and generate the filter output and error signal.

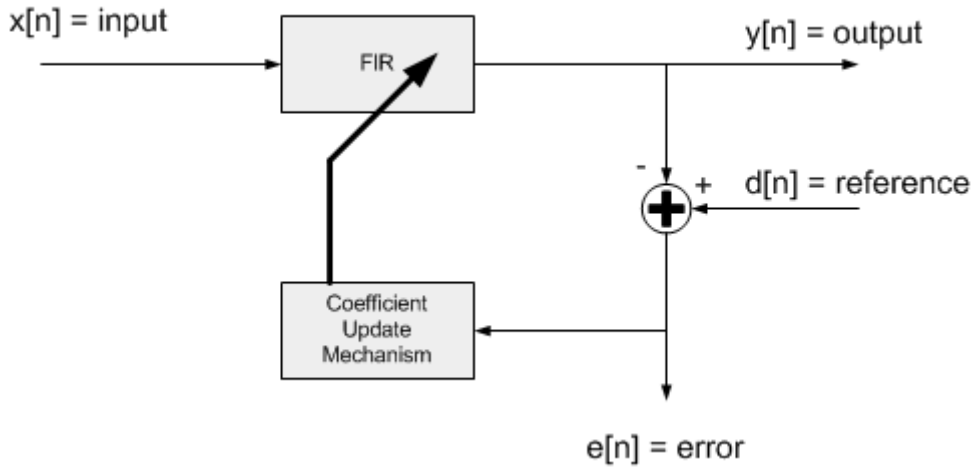


Figure6.11: Internal structure of the Least Mean Square filter

The functions operate on blocks of data and each call to the function processes blockSize samples through the filter. pSrc points to input signal, pRef points to reference signal, pOut points to output signal and pErr points to error signal. All arrays contain blockSize values.

The functions operate on a block-by-block basis. Internally, the filter coefficients b[n] are updated on a sample-by-sample basis. The convergence of the LMS filter is slower compared to the normalized LMS algorithm.

Algorithm:

The output signal y[n] is computed by a standard FIR filter:

```
y[n] = b[0] * x[n] + b[1] * x[n-1] + b[2] * x[n-2] + ...+ b[numTaps-1] * x[n-
↪numTaps+1]
```

The error signal equals the difference between the reference signal d[n] and the filter output:

```
e[n] = d[n] - y[n].
```

After each sample of the error signal is computed, the filter coefficients b[k] are updated on a sample-by-sample basis:

```
b[k] = b[k] + e[n] * mu * x[n-k],  for k=0, 1, ..., numTaps-1
```

where mu is the step size and controls the rate of coefficient convergence.

In the APIs, pCoeffs points to a coefficient array of size numTaps. Coefficients are stored in time reversed order.

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

pState points to a state array of size numTaps + blockSize - 1. Samples in the state buffer are stored in the order:

```
{x[n-numTaps+1], x[n-numTaps], x[n-numTaps-1], x[n-numTaps-2]....x[0], x[1],␣
↪..., x[blockSize-1]}
```

Note that the length of the state buffer exceeds the length of the coefficient array by blockSize-1 samples. The increased state buffer length allows circular addressing, which is traditionally used in FIR filters, to be avoided and yields a significant speed improvement. The state variables are updated after each block of data is processed.

Instance Structure

The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter and coefficient and state arrays cannot be shared among instances. There are separate instance structure declarations for each of the 3 supported data types.

Initialization Functions

There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numTaps, pCoeffs, mu, postShift (not for f32), pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros before static initialization. The code below statically initializes each of the 3 different data type filter instance structures

```
csi_lms_instance_f32 S = {numTaps, pState, pCoeffs, mu};
csi_lms_instance_q31 S = {numTaps, pState, pCoeffs, mu, postShift};
csi_lms_instance_q15 S = {numTaps, pState, pCoeffs, mu, postShift};
```

where numTaps is the number of filter coefficients in the filter; pState is the address of the state buffer; pCoeffs is the address of the coefficient buffer; mu is the step size parameter; and postShift is the shift applied to coefficients.

Fixed-Point Behavior:

Care must be taken when using the Q15 and Q31 versions of the LMS filter. The following issues must be considered:

- Scaling of coefficients
- Overflow and saturation

Scaling of Coefficients:

Filter coefficients are represented as fractional values and coefficients are restricted to lie in the range [-1 +1). The fixed-point functions have an additional scaling parameter postShift. At the output of the filter's accumulator is a shift register which shifts the result by postShift bits. This essentially scales the filter coefficients by 2^postShift and allows the filter coefficients to exceed the range [+1 -1). The value of postShift is set by the user based on the expected gain through the system being modeled.

Overflow and Saturation:

Overflow and saturation behavior of the fixed-point Q15 and Q31 versions are described separately as part of the function specific documentation below.

6.12.3 Function Docmentation

6.12.3.1 csi_lms_f32

```
void csi_lms_f32 (const csi_lms_instance_f32 *S, float32_t *pSrc, float32_t *pRef,␣
↪float32_t *pOut, float32_t *pErr, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the LMS structure.

`*pSrc`: points to the block of input data.

`*pRef`: points to the block of reference data.

`*pOut`: points to the block of output data.

`*pErr`: points to the block of error data.

`blockSize`: number of samples to process.

Returns:

none

6.12.3.2 csi_lms_q31

```
void csi_lms_q31 (const csi_lms_instance_q31 *S, q31_t *pSrc, q31_t *pRef, q31_t␣
↪*pOut, q31_t *pErr, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the LMS structure.

`*pSrc`: points to the block of input data.

`*pRef`: points to the block of reference data.

`*pOut`: points to the block of output data.

`*pErr`: points to the block of error data.

`blockSize`: number of samples to process.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clips. In order to avoid overflows completely the input signal must be scaled down by log2(numTaps) bits. The reference signal should not be scaled down. After all multiply-accumulates are performed, the 2.62 accumulator is shifted and saturated to 1.31 format to yield the final result. The output signal and error signal are in 1.31 format.

In this filter, filter coefficients are updated for each sample and the updation of filter cofficients are saturted.

6.12.3.3 csi_lms_q15

```
void csi_lms_q15 (const csi_lms_instance_q15 *S, q15_t *pSrc, q15_t *pRef, q15_t␣
↪*pOut, q15_t *pErr, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the LMS structure.

`*pSrc`: points to the block of input data.

`*pRef`: points to the block of reference data.

`*pOut`: points to the block of output data.

`*pErr`: points to the block of error data.

`blockSize`: number of samples to process.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. After all additions have been performed, the accumulator is truncated to 34.15 format by discarding low 15 bits. Lastly, the accumulator is saturated to yield a result in 1.15 format.

In this filter, filter coefficients are updated for each sample and the updation of filter cofficients are saturted.

6.12.3.4 csi_lms_init_f32

```
void csi_lms_init_f32 (csi_lms_instance_f32 *S, uint16_t numTaps, float32_t *pCoeffs,␣
↪float32_t *pState, float32_t mu, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the LMS structure.

`numTaps`: number of filter coefficients.

`*pCoeffs`: points to the coefficient buffer.

`*pState`: points to state buffer.

`mu`: step size that controls filter coefficient updates.

`blockSize`: number of samples to process.

Returns:

none

Description:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

The initial filter coefficients serve as a starting point for the adaptive filter. pState points to an array of length numTaps+blockSize-1 samples, where blockSize is the number of input samples processed by each call to *csi_lms_f32()*.

6.12.3.5 csi_lms_init_q31

```
void csi_lms_init_q31 (csi_lms_instance_q31 *S, uint16_t numTaps, q31_t *pCoeffs, q31_
↪t *pState, q31_t mu, uint32_t blockSize, uint32_t postShift)
```

Parameters:

`*S`: points to an instance of the LMS structure.

`numTaps`: number of filter coefficients.

`*pCoeffs`: points to the coefficient buffer.

`*pState`: points to state buffer.

`mu`: step size that controls filter coefficient updates.

`blockSize`: number of samples to process.

`postShift`: bit shift applied to coefficients.

Returns:

none

Description:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

The initial filter coefficients serve as a starting point for the adaptive filter. pState points to an array of length numTaps+blockSize-1 samples, where blockSize is the number of input samples processed by each call to *csi_lms_q31()*.

6.12.3.6 csi_lms_init_q15

```
void csi_lms_init_q15 (csi_lms_instance_q15 *S, uint16_t numTaps, q15_t *pCoeffs, q15_
↪t *pState, q15_t mu, uint32_t blockSize, uint32_t postShift)
```

Parameters:

`*S`: points to an instance of the LMS structure.

`numTaps`: number of filter coefficients.

`*pCoeffs`: points to the coefficient buffer.

`*pState`: points to state buffer.

`mu`: step size that controls filter coefficient updates.

`blockSize`: number of samples to process.

`postShift`: bit shift applied to coefficients.

Returns:

none

Description:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

The initial filter coefficients serve as a starting point for the adaptive filter. pState points to an array of length numTaps+blockSize-1 samples, where blockSize is the number of input samples processed by each call to *csi_lms_q15()*.

6.13 Normalized LMS Filters

6.13.1 Functions

- *csi_lms_norm_f32* : Processing function for the floating-point normalized LMS filter.
- *csi_lms_norm_q31* : Processing function for the Q31 normalized LMS filter.
- *csi_lms_norm_q15* : Processing function for the Q15 normalized LMS filter.
- *csi_lms_norm_init_f32* : Initialization function for the floating-point normalized LMS filter.
- *csi_lms_norm_init_q31* : Initialization function for the Q31 normalized LMS filter.
- *csi_lms_norm_init_q15* : Initialization function for the Q15 normalized LMS filter.

6.13.2 Description

This set of functions implements a commonly used adaptive filter. It is related to the Least Mean Square (LMS) adaptive filter and includes an additional normalization factor which increases the adaptation rate of the filter. The CSI DSP Library contains normalized LMS filter functions that operate on Q15, Q31, and floating-point data types.

A normalized least mean square (NLMS) filter consists of two components as shown below. The first component is a standard transversal or FIR filter. The second component is a coefficient update mechanism. The NLMS filter has two input signals. The "input" feeds the FIR filter while the "reference input" corresponds to the desired output of the FIR filter. That is, the FIR filter coefficients are updated so that the output of the FIR filter matches the reference input. The filter coefficient update mechanism is based on the difference between the FIR filter output and the reference input. This "error signal" tends towards zero as the filter adapts. The NLMS processing functions accept the input and reference input signals and generate the filter output and error signal.

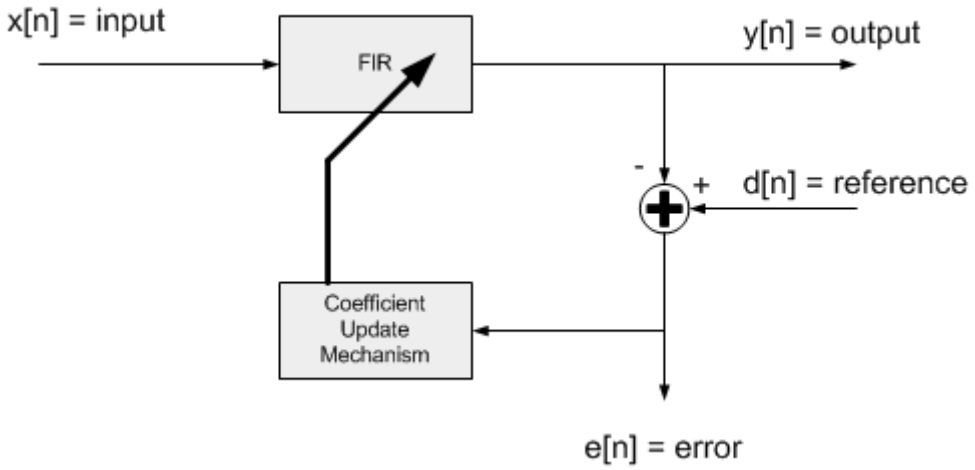


Figure6.12: Internal structure of the NLMS adaptive filter

The functions operate on blocks of data and each call to the function processes blockSize samples through the filter. pSrc points to input signal, pRef points to reference signal, pOut points to output signal and pErr points to error signal. All arrays contain blockSize values.

The functions operate on a block-by-block basis. Internally, the filter coefficients b[n] are updated on a sample-by-sample basis. The convergence of the LMS filter is slower compared to the normalized LMS algorithm.

Algorithm:

The output signal y[n] is computed by a standard FIR filter:

```
y[n] = b[0] * x[n] + b[1] * x[n-1] + b[2] * x[n-2] + ...+ b[numTaps-1] * x[n-
↪numTaps+1]
```

The error signal equals the difference between the reference signal d[n] and the filter output:

```
e[n] = d[n] - y[n].
```

After each sample of the error signal is computed the instanteous energy of the filter state variables is calculated:

```
E = x[n]^2 + x[n-1]^2 + ... + x[n-numTaps+1]^2.
```

The filter coefficients b[k] are then updated on a sample-by-sample basis:

```
b[k] = b[k] + e[n] * (mu/E) * x[n-k],  for k=0, 1, ..., numTaps-1
```

where mu is the step size and controls the rate of coefficient convergence.
In the APIs, pCoeffs points to a coefficient array of size numTaps. Coefficients are stored in time reversed order.

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

pState points to a state array of size numTaps + blockSize - 1. Samples in the state buffer are stored in the order:

```
{x[n-numTaps+1], x[n-numTaps], x[n-numTaps-1], x[n-numTaps-2]....x[0], x[1],␣
↪..., x[blockSize-1]}
```

Note that the length of the state buffer exceeds the length of the coefficient array by blockSize-1 samples. The increased state buffer length allows circular addressing, which is traditionally used in FIR filters, to be avoided and yields a significant speed improvement. The state variables are updated after each block of data is processed.

Instance Structure

The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter and coefficient and state arrays cannot be shared among instances. There are separate instance structure declarations for each of the 3 supported data types.

Initialization Functions

There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numTaps, pCoeffs, mu, energy, x0, pState. Also set all of the values in pState to zero. For Q7, Q15, and Q31 the following fields must also be initialized; recipTable, postShift

Instance structure cannot be placed into a const data section and it is recommended to use the initialization function.

Fixed-Point Behavior:

Care must be taken when using the Q15 and Q31 versions of the normalised LMS filter. The following issues must be considered:

- Scaling of coefficients
- Overflow and saturation

Scaling of Coefficients:

Filter coefficients are represented as fractional values and coefficients are restricted to lie in the range [-1 +1). The fixed-point functions have an additional scaling parameter postShift. At the output of the filter' s accumulator is a shift register which shifts the result by postShift bits. This essentially scales the filter coefficients by 2^postShift and allows the filter coefficients to exceed the range [+1 -1). The value of postShift is set by the user based on the expected gain through the system being modeled.

Overflow and Saturation:

Overflow and saturation behavior of the fixed-point Q15 and Q31 versions are described separately as part of the function specific documentation below.

6.13.3 Function Docmentation

6.13.3.1 csi_lms_norm_f32

```
void csi_lms_norm_f32 (csi_lms_norm_instance_f32 *S, float32_t *pSrc, float32_t *pRef,
↪ float32_t *pOut, float32_t *pErr, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the LMS structure.

`*pSrc`: points to the block of input data.

`*pRef`: points to the block of reference data.

`*pOut`: points to the block of output data.

`*pErr`: points to the block of error data.

`blockSize`: number of samples to process.

Returns:

none

6.13.3.2 csi_lms_norm_q31

```
void csi_lms_norm_q31 (csi_lms_norm_instance_q31 *S, q31_t *pSrc, q31_t *pRef, q31_t
↪*pOut, q31_t *pErr, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the LMS structure.

`*pSrc`: points to the block of input data.

`*pRef`: points to the block of reference data.

`*pOut`: points to the block of output data.

`*pErr`: points to the block of error data.

`blockSize`: number of samples to process.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clip. In order to avoid overflows completely the input signal must be scaled down by log2(numTaps) bits. The reference signal should not be scaled down. After all multiply-accumulates are performed, the 2.62 accumulator is shifted and saturated to 1.31 format to yield the final result. The output signal and error signal are in 1.31 format.

In this filter, filter coefficients are updated for each sample and the updation of filter cofficients are saturted.

6.13.3.3 csi_lms_norm_q15

```
void csi_lms_norm_q15 (csi_lms_norm_instance_q15 *S, q15_t *pSrc, q15_t *pRef, q15_t
↪*pOut, q15_t *pErr, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the LMS structure.

`*pSrc`: points to the block of input data.

`*pRef`: points to the block of reference data.

`*pOut`: points to the block of output data.

`*pErr`: points to the block of error data.

`blockSize`: number of samples to process.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. After all additions have been performed, the accumulator is truncated to 34.15 format by discarding low 15 bits. Lastly, the accumulator is saturated to yield a result in 1.15 format.

In this filter, filter coefficients are updated for each sample and the updation of filter cofficients are saturted.

6.13.3.4 csi_lms_norm_init_f32

```
void csi_lms_norm_init_f32 (csi_lms_norm_instance_f32 *S, uint16_t numTaps, float32_t␣
↪*pCoeffs, float32_t *pState, float32_t mu, uint32_t blockSize)
```

Parameters:

`*S`: points to an instance of the LMS structure.

`numTaps`: number of filter coefficients.

`*pCoeffs`: points to coefficient buffer.

`*pState`: points to state buffer.

`mu`: step size that controls filter coefficient updates.

`blockSize`: number of samples to process.

Returns:

none

Scaling and Overflow Behavior:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

The initial filter coefficients serve as a starting point for the adaptive filter. pState points to an array of length numTaps+blockSize-1 samples, where blockSize is the number of input samples processed by each call to *csi_lms_norm_f32()*.

6.13.3.5 csi_lms_norm_init_q31

```
void csi_lms_norm_init_q31 (csi_lms_norm_instance_q31 *S, uint16_t numTaps, q31_t␣
↪*pCoeffs, q31_t *pState, q31_t mu, uint32_t blockSize, uint8_t postShift)
```

Parameters:

`*S`: points to an instance of the LMS structure.

`numTaps`: number of filter coefficients.

*pCoeffs: points to coefficient buffer.

*pState: points to state buffer.

mu: step size that controls filter coefficient updates.

blockSize: number of samples to process.

postShift: number of samples to process.

Returns:

none

Scaling and Overflow Behavior:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

The initial filter coefficients serve as a starting point for the adaptive filter. pState points to an array of length numTaps+blockSize-1 samples, where blockSize is the number of input samples processed by each call to *csi_lms_norm_q31()*.

6.13.3.6 csi_lms_norm_init_q15

```
void csi_lms_norm_init_q15 (csi_lms_norm_instance_q15 *S, uint16_t numTaps, q15_t␣
↪*pCoeffs, q15_t *pState, q15_t mu, uint32_t blockSize, uint8_t postShift)
```

Parameters:

*S: points to an instance of the LMS structure.

numTaps: number of filter coefficients.

*pCoeffs: points to coefficient buffer.

*pState: points to state buffer.

mu: step size that controls filter coefficient updates.

blockSize: number of samples to process.

postShift: number of samples to process.

Returns:

none

Scaling and Overflow Behavior:

pCoeffs points to the array of filter coefficients stored in time reversed order:

```
{b[numTaps-1], b[numTaps-2], b[N-2], ..., b[1], b[0]}
```

The initial filter coefficients serve as a starting point for the adaptive filter. pState points to an array of length numTaps+blockSize-1 samples, where blockSize is the number of input samples processed by each call to *csi_lms_norm_q15()*.

CHAPTER 7

Interpolation Functions

7.1 Linear Interpolation

7.1.1 Functions

- *csi_linear_interp_f32* : Process function for the floating-point Linear Interpolation Function.
- *csi_linear_interp_q31* : Process function for the Q31 Linear Interpolation Function.
- *csi_linear_interp_q15* : Process function for the Q15 Linear Interpolation Function.
- *csi_linear_interp_q7* : Process function for the Q7 Linear Interpolation Function.

7.1.2 Description

Linear interpolation is a method of curve fitting using linear polynomials. Linear interpolation works by effectively drawing a straight line between two neighboring samples and returning the appropriate point along that line

A Linear Interpolate function calculates an output value(y), for the input(x) using linear interpolation of the input values x0, x1(nearest input values) and the output values y0 and y1(nearest output values)

Algorithm:

```
y = y0 + (x - x0) * ((y1 - y0)/(x1-x0))
where x0, x1 are nearest values of input x
 y0, y1 are nearest values to output y
```

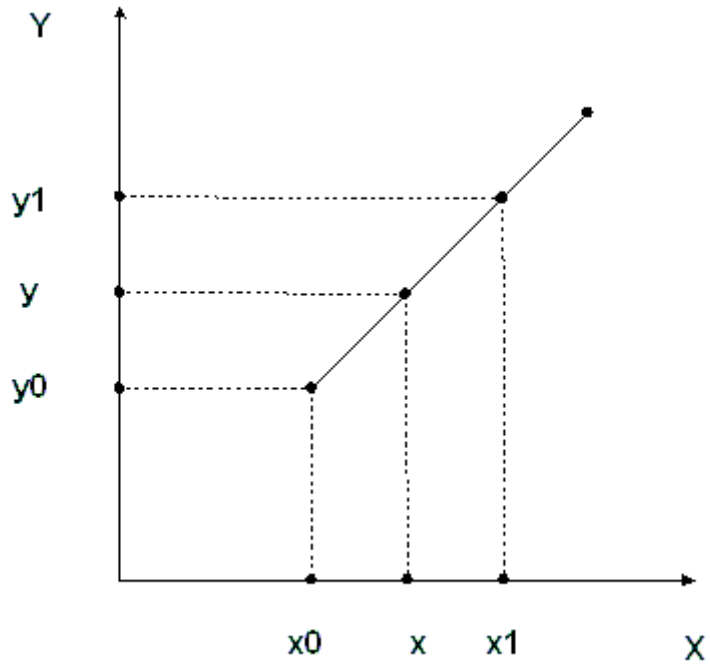


Figure7.1: Linear interpolation

This set of functions implements Linear interpolation process for Q7, Q15, Q31, and floating-point data types. The functions operate on a single sample of data and each call to the function returns a single processed value. S points to an instance of the Linear Interpolate function data structure. x is the input sample value. The functions returns the output value.

if x is outside of the table boundary, Linear interpolation returns first value of the table if x is below input range and returns last value of table if x is above range.

7.1.3 Function Docmentation

7.1.3.1 csi_linear_interp_f32

```
float32_t csi_linear_interp_f32 (csi_linear_interp_instance_f32 *S, float32_t x)
```

Parameters:

`*S`: points to an instance of the Linear Interpolation structure.

`x`: input sample to process.

Returns:

y processed output sample.

7.1.3.2 csi_linear_interp_q31

```
q31_t  csi_linear_interp_q31 (q31_t *pYData, q31_t x, uint32_t nValues)
```

Parameters:

`*pYData`: points to an instance of the Linear Interpolation structure.

`x`: input sample to process.

`nValues`: number of table values.

Returns:

y processed output sample.

Description:

Input sample x is in 12.20 format which contains 12 bits for table index and 20 bits for fractional part. This function can support maximum of table size 2^12.

7.1.3.3 csi_linear_interp_q15

```
q15_t  csi_linear_interp_q15 (q15_t *pYData, q31_t x, uint32_t nValues)
```

Parameters:

`*pYData`: points to an instance of the Linear Interpolation structure.

`x`: input sample to process.

`nValues`: number of table values.

Returns:

y processed output sample.

Description:

Input sample x is in 12.20 format which contains 12 bits for table index and 20 bits for fractional part. This function can support maximum of table size 2^12.

7.1.3.4 csi_linear_interp_q7

```
q7_t csi_linear_interp_q7 (q7_t *pYData, q31_t x, uint32_t nValues)
```

Parameters:

`*pYData`: points to an instance of the Linear Interpolation structure.

`x`: input sample to process.

`nValues`: number of table values.

Returns:

y processed output sample.

Description:

Input sample x is in 12.20 format which contains 12 bits for table index and 20 bits for fractional part. This function can support maximum of table size 2^12.

7.2 Bilinear Interpolation

7.2.1 Functions

- *csi_bilinear_interp_f32* : Process function for the floating-point bilinear Interpolation Function.
- *csi_bilinear_interp_q31* : Process function for the Q31 bilinear Interpolation Function.
- *csi_bilinear_interp_q15* : Process function for the Q15 bilinear Interpolation Function.
- *csi_bilinear_interp_q7* : Process function for the Q7 Linear Interpolation Function.

7.2.2 Description

Bilinear interpolation is an extension of linear interpolation applied to a two dimensional grid. The underlying function f(x, y) is sampled on a regular grid and the interpolation process determines values between the grid points. Bilinear interpolation is equivalent to two step linear interpolation, first in the x-dimension and then in the y-dimension. Bilinear interpolation is often used in image processing to rescale images. The CSI DSP library provides bilinear interpolation functions for Q7, Q15, Q31, and floating-point data types.

Algorithm

The instance structure used by the bilinear interpolation functions describes a two dimensional data table. For floating-point, the instance structure is defined as:

```
typedef struct
{
  uint16_t numRows;
  uint16_t numCols;
  float32_t *pData;
} csi_bilinear_interp_instance_f32;
```

where numRows specifies the number of rows in the table; numCols specifies the number of columns in the table; and pData points to an array of size numRows*numCols values. The data table pTable is organized in row order and the supplied data values fall on integer indexes. That is, table element (x,y) is located at pTable[x + y*numCols] where x and y are integers.

Let (x, y) specify the desired interpolation point. Then define:

```
XF = floor(x)
YF = floor(y)
```

The interpolated output point is computed as:

```
 f(x, y) = f(XF, YF) * (1-(x-XF)) * (1-(y-YF))
+ f(XF+1, YF) * (x-XF)*(1-(y-YF))
+ f(XF, YF+1) * (1-(x-XF))*(y-YF)
+ f(XF+1, YF+1) * (x-XF)*(y-YF)
```

Note that the coordinates (x, y) contain integer and fractional components. The integer components specify which portion of the table to use while the fractional components control the interpolation processor.

if (x,y) are outside of the table boundary, Bilinear interpolation returns zero output.

7.2.3 Function Docmentation

7.2.3.1 csi_bilinear_interp_f32

```
float32_t csi_bilinear_interp_f32 (const csi_bilinear_interp_instance_f32 *S, float32_
↪t X, float32_t Y)
```

Parameters:

`*S`: points to an instance of the Interpolation structure.

`X`: input sample to process.

`Y`: input sample to process.

Returns:

out interpolated value..

7.2.3.2 csi_bilinear_interp_q31

```
q31_t  csi_bilinear_interp_q31 (csi_bilinear_interp_instance_q31 *S, q31_t X, q31_t Y)
```

Parameters:

`*pYData`: points to an instance of the Interpolation structure.

`X`: interpolation coordinate in 12.20 format.

`Y`: interpolation coordinate in 12.20 format.

Returns:

out interpolated value.

7.2.3.3 csi_bilinear_interp_q15

```
q15_t  csi_bilinear_interp_q15 (csi_bilinear_interp_instance_q15 *S, q31_t X, q31_t Y)
```

Parameters:

`*pYData`: points to an instance of the Interpolation structure.

`X`: interpolation coordinate in 12.20 format.

`Y`: interpolation coordinate in 12.20 format.

Returns:

out interpolated value.

7.2.3.4 csi_bilinear_interp_q7

```
q7_t csi_bilinear_interp_q7 (csi_bilinear_interp_instance_q7 *S, q31_t X, q31_t Y)
```

Parameters:

`*pYData`: points to an instance of the Interpolation structure.

`X`: interpolation coordinate in 12.20 format.

`Y`: interpolation coordinate in 12.20 format.

Returns:

out interpolated value.

CHAPTER 8

Matrix Functions

Description

This set of functions provides basic matrix math operations. The functions operate on matrix data structures. For example, the type definition for the floating-point matrix structure is shown below:

```
typedef struct
{
  uint16_t numRows;     // number of rows of the matrix.
  uint16_t numCols;     // number of columns of the matrix.
  float32_t *pData;     // points to the data of the matrix.
} csi_matrix_instance_f32;
```

There are similar definitions for Q15 and Q31 data types.

The structure specifies the size of the matrix and then points to an array of data. The array is of size numRows X numCols and the values are arranged in row order. That is, the matrix element (i, j) is stored at:

```
pData[i*numCols + j]
```

Init Functions

There is an associated initialization function for each type of matrix data structure. The initialization function sets the values of the internal structure fields. Refer to the function csi_mat_init_f32(), csi_mat_init_q31() and csi_mat_init_q15() for floating-point, Q31 and Q15 types, respectively.

Use of the initialization function is optional. However, if initialization function is used then the instance structure cannot be placed into a const data section. To place the instance structure in a const data section, manually initialize the data structure. For example:

```
csi_matrix_instance_f32 S = {nRows, nColumns, pData};
csi_matrix_instance_q31 S = {nRows, nColumns, pData};
csi_matrix_instance_q15 S = {nRows, nColumns, pData};
```

where nRows specifies the number of rows, nColumns specifies the number of columns, and pData points to the data array.

8.1 Matrix Initialization

8.1.1 Functions

- *csi_mat_init_f32* : Floating-point matrix initialization.
- *csi_mat_init_q31* : Q31 matrix initialization.
- *csi_mat_init_q15* : Q15 matrix initialization.

8.1.2 Description

Initializes the underlying matrix data structure. The functions set the numRows, numCols, and pData fields of the matrix data structure.

8.1.3 Function Docmentation

8.1.3.1 csi_mat_init_f32

```
void csi_mat_init_f32 (csi_matrix_instance_f32 *S, uint16_t nRows, uint16_t nColumns,␣
↪float32_t *pData)
```

Parameters:

`*S`: points to an instance of matrix structure.

`nRows`: number of rows in the matrix.

`nColumns`: number of columns in the matrix.

`*pData`: points to the matrix data array.

Returns:

none

8.1.3.2 csi_mat_init_q31

```
void csi_mat_init_q31 (csi_matrix_instance_q31 *S, uint16_t nRows, uint16_t nColumns,␣
↪q31_t *pData)
```

Parameters:

`*S`: points to an instance of matrix structure.

`nRows`: number of rows in the matrix.

`nColumns`: number of columns in the matrix.

`*pData`: points to the matrix data array.

Returns:

none

8.1.3.3 csi_mat_init_q15

```
void csi_mat_init_q15 (csi_matrix_instance_q15 *S, uint16_t nRows, uint16_t nColumns,␣
↪q15_t *pData)
```

Parameters:

`*S`: points to an instance of matrix structure.

`nRows`: number of rows in the matrix.

`nColumns`: number of columns in the matrix.

`*pData`: points to the matrix data array.

Returns:

none

8.2 Matrix Addition

8.2.1 Functions

- *csi_mat_add_f32* : Floating-point matrix addition.
- *csi_mat_add_q31* : Q31 matrix addition.
- *csi_mat_add_q15* : Q15 matrix addition.

8.2.2 Description

Adds two matrices.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{bmatrix}$$

Figure8.1: Addition of two 3 x 3 matrices

The functions check to make sure that pSrcA, pSrcB, and pDst have the same number of rows and columns.

8.2.3 Function Docmentation

8.2.3.1 csi_mat_add_f32

```
csi_status csi_mat_add_f32 (const csi_matrix_instance_f32 *pSrcA, const csi_matrix_
↪instance_f32 *pSrcB, csi_matrix_instance_f32 *pDst)
```

Parameters:

`*pSrcA`: points to the first input matrix structure

`*pSrcB`: points to the second input matrix structure.

`*pDst`: points to output matrix structure.

Returns:

none

8.2.3.2 csi_mat_add_q31

```
csi_status csi_mat_add_q31 (const csi_matrix_instance_q31 *pSrcA, const csi_matrix_
↪instance_q31 *pSrcB, csi_matrix_instance_q31 *pDst)
```

Parameters:

`*pSrcA`: points to the first input matrix structure

`*pSrcB`: points to the second input matrix structure.

`*pDst`: points to output matrix structure.

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q31 range [0x80000000 0x7FFFFFFF] will be saturated.

8.2.3.3 csi_mat_add_q15

```
csi_status csi_mat_add_q15 (const csi_matrix_instance_q15 *pSrcA, const csi_matrix_
↪instance_q15 *pSrcB, csi_matrix_instance_q15 *pDst)
```

Parameters:

`*pSrcA`: points to the first input matrix structure

`*pSrcB`: points to the second input matrix structure.

`*pDst`: points to output matrix structure.

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] will be saturated.

8.3 Complex Matrix Multiplication

8.3.1 Functions

- *csi_mat_cmplx_mult_f32* : Floating-point Complex matrix addition.
- *csi_mat_cmplx_mult_q31* : Q31 Complex matrix addition.
- *csi_mat_cmplx_mult_q15* : Q15 Complex matrix addition.

8.3.2 Description

Complex Matrix multiplication is only defined if the number of columns of the first matrix equals the number of rows of the second matrix. Multiplying an M x N matrix with an N x P matrix results in an M x P matrix. When matrix size checking is enabled, the functions check:

1. that the inner dimensions of pSrcA and pSrcB are equal.
2. that the size of the output matrix equals the outer dimensions of pSrcA and pSrcB.

8.3.3 Function Docmentation

8.3.3.1 csi_mat_cmplx_mult_f32

```
csi_status csi_mat_cmplx_mult_f32 (const csi_matrix_instance_f32 *pSrcA, const csi_
↪matrix_instance_f32 *pSrcB, csi_matrix_instance_f32 *pDst)
```

Parameters:

`*pSrcA`: points to the first input matrix structure

`*pSrcB`: points to the second input matrix structure.

`*pDst`: points to output matrix structure.

Returns:

none

8.3.3.2 csi_mat_cmplx_mult_q31

```
csi_status csi_mat_cmplx_mult_q31 (const csi_matrix_instance_q31 *pSrcA, const csi_
↪matrix_instance_q31 *pSrcB, csi_matrix_instance_q31 *pDst)
```

Parameters:

`*pSrcA`: points to the first input matrix structure

`*pSrcB`: points to the second input matrix structure.

`*pDst`: points to output matrix structure.

Returns:

none The function returns either CSKY_MATH_SIZE_MISMATCH or CSKY_MATH_SUCCESS based on the outcome of size checking.

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid intermediate overflows. The input is thus scaled down by log2(numColsA) bits to avoid overflows, as a total of numColsA additions are performed internally. The 2.62 accumulator is right shifted by 31 bits and saturated to 1.31 format to yield the final result.

8.3.3.3 csi_mat_cmplx_mult_q15

```
csi_status csi_mat_cmplx_mult_q15 (const csi_matrix_instance_q15 *pSrcA, const csi_
↪matrix_instance_q15 *pSrcB, csi_matrix_instance_q15 *pDst, q15_t *pScratch)
```

Parameters:

`*pSrcA`: points to the first input matrix structure

`*pSrcB`: points to the second input matrix structure.

`*pDst`: points to output matrix structure.

`*pScratch`: points to the array for storing intermediate results

Returns:

none

Conditions for optimum performance

Input, output and state buffers should be aligned by 32-bit

Restrictions

If the silicon does not support unaligned memory access enable the macro UNALIGNED_SUPPORT_DISABLE In this case input, output, scratch buffers should be aligned by 32-bit

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. The inputs to the multiplications are in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. This approach provides 33 guard bits and there is no risk of overflow. The 34.30 result is then truncated to 34.15 format by discarding the low 15 bits and then saturated to 1.15 format.

Refer to *csi_mat_mult_fast_q15()* for a faster but less precise version of this function.

8.4 Matrix Inverse

8.4.1 Functions

- *csi_mat_inverse_f32* : Floating-point matrix inverse.
- *csi_mat_inverse_f64* : Floating-point matrix inverse.

8.4.2 Description

Computes the inverse of a matrix, which is not support for I805.

The inverse is defined only if the input matrix is square and non-singular (the determinant is non-zero). The function checks that the input and output matrices are square and of the same size.

Matrix inversion is numerically sensitive and the CSI DSP library only supports matrix inversion of floating-point matrices.

Algorithm

The Gauss-Jordan method is used to find the inverse. The algorithm performs a sequence of elementary row-operations until it reduces the input matrix to an identity matrix. Applying the same sequence of elementary row-operations to an identity matrix yields the inverse matrix. If the input matrix is singular, then the algorithm terminates and returns error status CSKY_MATH_SINGULAR.

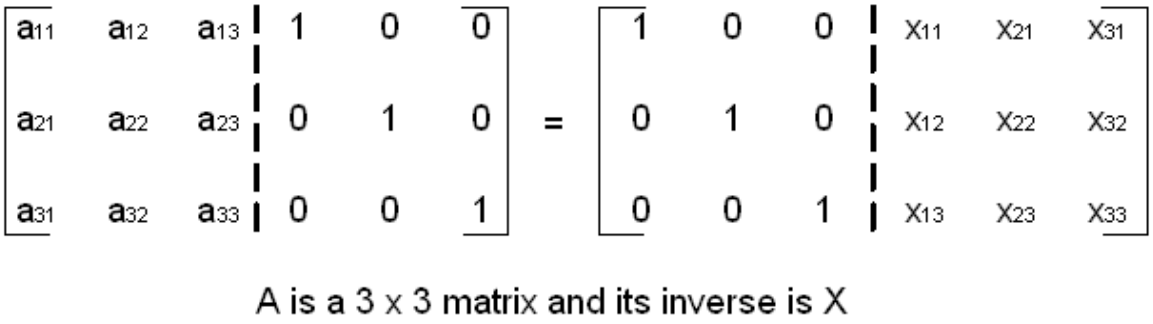


Figure8.2: Matrix Inverse of a 3 x 3 matrix using Gauss-Jordan Method

8.4.3 Function Docmentation

8.4.3.1 csi_mat_inverse_f32

```
csi_status csi_mat_inverse_f32 (const csi_matrix_instance_f32 *pSrc, csi_matrix_
↪instance_f32 *pDst)
```

Parameters:

*pSrc: points to input matrix structure

*pDst: points to output matrix structure.

Returns:

none

8.4.3.2 csi_mat_inverse_f64

```
csi_status csi_mat_inverse_f64 (const csi_matrix_instance_f64 *pSrc, csi_matrix_
↪instance_f64 *pDst)
```

Parameters:

*pSrc: points to input matrix structure

*pDst: points to output matrix structure.

Returns:

none

8.5 Matrix Multiplication

8.5.1 Functions

- *csi_mat_mult_f32* : Floating-point matrix multiplication.
- *csi_mat_mult_q31* : Q31 matrix multiplication.
- *csi_mat_mult_q15* : Q15 matrix multiplication.
- *csi_mat_mult_fast_q31* : Q31 matrix multiplication.
- *csi_mat_mult_fast_q15* : Q15 matrix multiplication.

8.5.2 Description

Multiplies two matrices, the fast version is not support for I805/C860.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}xb_{11}+a_{12}xb_{21}+a_{13}xb_{31} & a_{11}xb_{12}+a_{12}xb_{22}+a_{13}xb_{32} & a_{11}xb_{13}+a_{12}xb_{23}+a_{13}xb_{33} \\ a_{21}xb_{11}+a_{22}xb_{21}+a_{23}xb_{31} & a_{21}xb_{12}+a_{22}xb_{22}+a_{23}xb_{32} & a_{21}xb_{13}+a_{22}xb_{23}+a_{23}xb_{33} \\ a_{31}xb_{11}+a_{32}xb_{21}+a_{33}xb_{31} & a_{31}xb_{12}+a_{32}xb_{22}+a_{33}xb_{32} & a_{31}xb_{13}+a_{32}xb_{23}+a_{33}xb_{33} \end{bmatrix}$$

Figure8.3: Multiplication of two 3 x 3 matrices

Matrix multiplication is only defined if the number of columns of the first matrix equals the number of rows of the second matrix. Multiplying an M x N matrix with an N x P matrix results in an M x P matrix. When matrix size checking is enabled, the functions check:

1. that the inner dimensions of pSrcA and pSrcB are equal.
2. that the size of the output matrix equals the outer dimensions of pSrcA and pSrcB.

8.5.3 Function Docmentation

8.5.3.1 csi_mat_mult_f32

```
csi_status csi_mat_mult_f32 (const csi_matrix_instance_f32 *pSrcA, const csi_matrix_
↪instance_f32 *pSrcB, csi_matrix_instance_f32 *pDst)
```

Parameters:

`*pSrcA`: points to input matrix structure.

`*pSrcB`: points to input matrix structure.

`*pDst`: points to output matrix structure.

Returns:

none

8.5.3.2 csi_mat_mult_q31

```
csi_status csi_mat_mult_q31 (const csi_matrix_instance_q31 *pSrcA, const csi_matrix_
↪instance_q31 *pSrcB, csi_matrix_instance_q31 *pDst)
```

Parameters:

`*pSrcA`: points to input matrix structure.

`*pSrcB`: points to input matrix structure.

`*pDst`: points to output matrix structure.

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid intermediate overflows. The input is thus scaled down by log2(numColsA) bits to avoid overflows, as a total of numColsA additions are performed internally. The 2.62 accumulator is right shifted by 31 bits and saturated to 1.31 format to yield the final result.

8.5.3.3 csi_mat_mult_q15

```
csi_status csi_mat_mult_q15 (const csi_matrix_instance_q15 *pSrcA, const csi_matrix_
↪instance_q15 *pSrcB, csi_matrix_instance_q15 *pDst, q15_t *pState)
```

Parameters:

`*pSrcA`: points to input matrix structure.

`*pSrcB`: points to input matrix structure.

`*pDst`: points to output matrix structure.

`*pState`: points to the array for storing intermediate results (Unused)

Returns:

none

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. The inputs to the multiplications are in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. This approach provides 33 guard bits and there is no risk of overflow. The 34.30 result is then truncated to 34.15 format by discarding the low 15 bits and then saturated to 1.15 format.

8.5.3.4 csi_mat_mult_fast_q31

```
csi_status csi_mat_mult_fast_q31 (const csi_matrix_instance_q31 *pSrcA, const csi_
↪matrix_instance_q31 *pSrcB, csi_matrix_instance_q31 *pDst)
```

Parameters:

`*pSrcA`: points to input matrix structure.

`*pSrcB`: points to input matrix structure.

`*pDst`: points to output matrix structure.

Returns:

none

Scaling and Overflow Behavior:

The difference between the function *csi_mat_mult_q31()* and this fast variant is that the fast variant use a 32-bit rather than a 64-bit accumulator. The result of each 1.31 x 1.31 multiplication is truncated to 2.30 format. These intermediate results are accumulated in a 32-bit register in 2.30 format. Finally, the accumulator is saturated and converted to a 1.31 result.

The fast version has the same overflow behavior as the standard version but provides less precision since it discards the low 32 bits of each multiplication result. In order to avoid overflows completely the input signals must be scaled down. Scale down one of the input matrices by log2(numColsA) bits to avoid overflows, as a total of numColsA additions are computed internally for each output element.

See *csi_mat_mult_q31()* for a slower implementation of this function which uses 64-bit accumulation to provide higher precision.

8.5.3.5 csi_mat_mult_fast_q15

```
csi_status csi_mat_mult_fast_q15 (const csi_matrix_instance_q15 *pSrcA, const csi_
↪matrix_instance_q15 *pSrcB, csi_matrix_instance_q15 *pDst, q15_t *pState)
```

Parameters:

`*pSrcA`: points to input matrix structure.

`*pSrcB`: points to input matrix structure.

`*pDst`: points to output matrix structure.

`*pState`: points to the array for storing intermediate results (Unused)

Returns:

none

Scaling and Overflow Behavior:

The difference between the function *csi_mat_mult_q15()* and this fast variant is that the fast variant use a 32-bit rather than a 64-bit accumulator. The result of each 1.15 x 1.15 multiplication is truncated to 2.30 format. These intermediate results are accumulated in a 32-bit register in 2.30 format. Finally, the accumulator is saturated and converted to a 1.15 result.

The fast version has the same overflow behavior as the standard version but provides less precision since it discards the low 16 bits of each multiplication result. In order to avoid overflows completely the input signals must be scaled down. Scale down one of the input matrices by log2(numColsA) bits to avoid overflows, as a total of numColsA additions are computed internally for each output element.

See *csi_mat_mult_q15()* for a slower implementation of this function which uses 64-bit accumulation to provide higher precision.

8.6 Matrix Scale

8.6.1 Functions

- *csi_mat_scale_f32* : Floating-point matrix scaling.
- *csi_mat_scale_q31* : Q31 matrix scaling.
- *csi_mat_scale_q15* : Q15 matrix scaling.

8.6.2 Description

Multiplies a matrix by a scalar. This is accomplished by multiplying each element in the matrix by the scalar. For example:

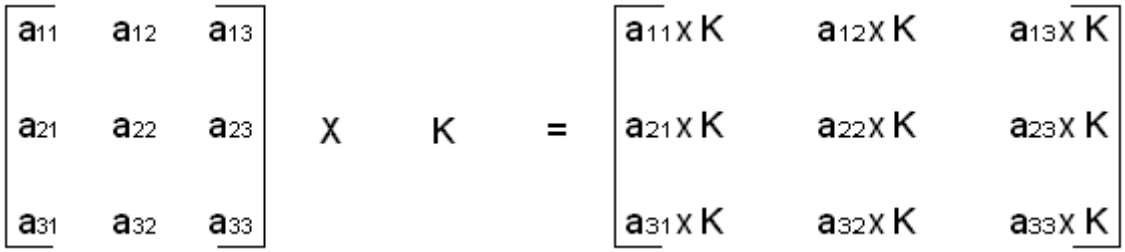


Figure8.4: Matrix Scaling of a 3 x 3 matrix

The function checks to make sure that the input and output matrices are of the same size.

In the fixed-point Q15 and Q31 functions, scale is represented by a fractional multiplication scaleFract and an arithmetic shift shift. The shift allows the gain of the scaling operation to exceed 1.0. The overall scale factor applied to the fixed-point data is

```
scale = scaleFract * 2^shift.
```

8.6.3 Function Docmentation

8.6.3.1 csi_mat_scale_f32

```
csi_status csi_mat_scale_f32 (const csi_matrix_instance_f32 *pSrc, float32_t scale,␣
↪csi_matrix_instance_f32 *pDst)
```

Parameters:

`*pSrc`: points to input matrix structure.

`scale`: scale factor to be applied.

`*pDst`: points to output matrix structure.

Returns:

none

8.6.3.2 csi_mat_scale_q31

```
csi_status csi_mat_scale_q31 (const csi_matrix_instance_q31 *pSrc, q31_t scaleFract,␣
↪int32_t shift, csi_matrix_instance_q31 *pDst)
```

Parameters:

`*pSrc`: points to input matrix structure.

`scaleFract`: fractional portion of the scale factor.

`shift`: number of bits to shift the result by.

`*pDst`: points to output matrix structure.

Returns:

none

Scaling and Overflow Behavior:

The input data pSrc and scaleFract are in 1.31 format. These are multiplied to yield a 2.62 intermediate result and this is shifted with saturation to 1.31 format.

8.6.3.3 csi_mat_scale_q15

```
csi_status csi_mat_scale_q15 (const csi_matrix_instance_q15 *pSrc, q15_t scaleFract,␣
↪int32_t shift, csi_matrix_instance_q15 *pDst)
```

Parameters:

`*pSrc`: points to input matrix structure.

`scaleFract`: fractional portion of the scale factor.

`shift`: number of bits to shift the result by.

`*pDst`: points to output matrix structure.

Returns:

none

Scaling and Overflow Behavior:

The input data pSrc and scaleFract are in 1.15 format. These are multiplied to yield a 2.30 intermediate result and this is shifted with saturation to 1.15 format.

8.7 Matrix Subtraction

8.7.1 Functions

- *csi_mat_sub_f32* : Floating-point matrix subtraction.
- *csi_mat_sub_q31* : Q31 matrix subtraction.
- *csi_mat_sub_q15* : Q15 matrix subtraction.

8.7.2 Description

Subtract two matrices.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} - \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}-b_{11} & a_{12}-b_{12} & a_{13}-b_{13} \\ a_{21}-b_{21} & a_{22}-b_{22} & a_{23}-b_{23} \\ a_{31}-b_{31} & a_{32}-b_{32} & a_{33}-b_{33} \end{bmatrix}$$

Figure8.5: Subraction of two 3 x 3 matrices

The functions check to make sure that pSrcA, pSrcB, and pDst have the same number of rows and columns.

8.7.3 Function Docmentation

8.7.3.1 csi_mat_sub_f32

```
csi_status csi_mat_sub_f32 (const csi_matrix_instance_f32 *pSrcA, const csi_matrix_
↪instance_f32 *pSrcB, csi_matrix_instance_f32 *pDst)
```

Parameters:

`*pSrcA`: points to the first input matrix structure

`*pSrcB`: points to the second input matrix structure.

`*pDst`: points to output matrix structure.

Returns:

none

8.7.3.2 csi_mat_sub_q31

```
csi_status csi_mat_sub_q31 (const csi_matrix_instance_q31 *pSrcA, const csi_matrix_
↪instance_q31 *pSrcB, csi_matrix_instance_q31 *pDst)
```

Parameters:

`*pSrcA`: points to the first input matrix structure

`*pSrcB`: points to the second input matrix structure.

`*pDst`: points to output matrix structure.

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q31 range [0x80000000 0x7FFFFFFF] will be saturated.

8.7.3.3 csi_mat_sub_q15

```
csi_status csi_mat_sub_q15 (const csi_matrix_instance_q15 *pSrcA, const csi_matrix_
↪instance_q15 *pSrcB, csi_matrix_instance_q15 *pDst)
```

Parameters:

`*pSrcA`: points to the first input matrix structure

`*pSrcB`: points to the second input matrix structure.

`*pDst`: points to output matrix structure.

Returns:

none

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] will be saturated.

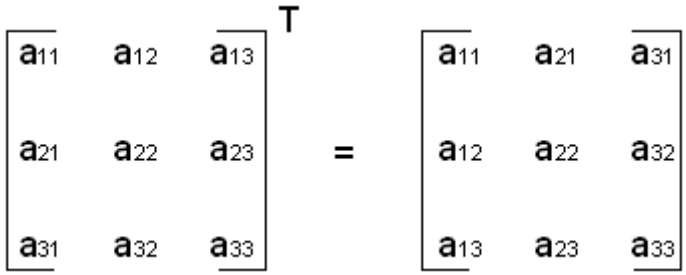
8.8 Matrix Transpose

8.8.1 Functions

- *csi_mat_trans_f32* : Floating-point matrix subtraction.
- *csi_mat_trans_q31* : Q31 matrix subtraction.
- *csi_mat_trans_q15* : Q15 matrix subtraction.

8.8.2 Description

Tranposes a matrix. Transposing an M x N matrix flips it around the center diagonal and results in an N x M matrix.



Transpose of a 3 x 3 matrix

8.8.3 Function Docmentation

8.8.3.1 csi_mat_trans_f32

```
csi_status csi_mat_trans_f32 (const csi_matrix_instance_f32 *pSrc, csi_matrix_
↪instance_f32 *pDst)
```

Parameters:

`*pSrc`: points to the input matrix structure

`*pDst`: points to output matrix structure.

Returns:

none

8.8.3.2 csi_mat_trans_q31

```
csi_status csi_mat_trans_q31 (const csi_matrix_instance_q31 *pSrc, csi_matrix_
↪instance_q31 *pDst)
```

Parameters:

`*pSrc`: points to the input matrix structure

`*pDst`: points to output matrix structure.

Returns:

none

8.8.3.3 csi_mat_trans_q15

```
csi_status csi_mat_trans_q15 (const csi_matrix_instance_q15 *pSrc, csi_matrix_
↪instance_q15 *pDst)
```

Parameters:

`*pSrc`: points to the input matrix structure

`*pDst`: points to output matrix structure.

Returns:

none

CHAPTER 9

Statistics Functions

9.1 Maximum

9.1.1 Functions

- *csi_max_f32* : Maximum value of a floating-point vector.
- *csi_max_q31* : Maximum value of a Q31 vector.
- *csi_max_q15* : Maximum value of a Q15 vector.
- *csi_max_q7* : Maximum value of a Q7 vector.

9.1.2 Description

Computes the maximum value of an array of data. The function returns both the maximum value and its position within the array. There are separate functions for floating-point, Q31, Q15, and Q7 data types.

9.1.3 Function Docmentation

9.1.3.1 csi_max_f32

```
void csi_max_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult, uint32_t␣
↪*pIndex)
```

Parameters:

*pSrc: points to the input vector.

blockSize: length of the input vector.

*pResult: maximum value returned here.

*pIndex: index of maximum value returned here.

Returns:

none.

9.1.3.2 csi_max_q31

```
void csi_max_q31 (q31_t *pSrc, uint32_t blockSize, q31_t *pResult, uint32_t *pIndex)
```

Parameters:

*pSrc: points to the input vector.

blockSize: length of the input vector.

*pResult: maximum value returned here.

*pIndex: index of maximum value returned here.

Returns:

none.

9.1.3.3 csi_max_q15

```
void csi_max_q15 (q15_t *pSrc, uint32_t blockSize, q15_t *pResult, uint16_t *pIndex)
```

Parameters:

*pSrc: points to the input vector.

blockSize: length of the input vector.

*pResult: maximum value returned here.

*pIndex: index of maximum value returned here.

Returns:

none.

9.1.3.4 csi_max_q7

```
void csi_max_q7 (q7_t *pSrc, uint32_t blockSize, q7_t *pResult, uint16_t *pIndex)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: maximum value returned here.

`*pIndex`: index of maximum value returned here.

Returns:

none.

9.2 Minimum

9.2.1 Functions

- *csi_min_f32* : Minimum value of a floating-point vector.
- *csi_min_q31* : Minimum value of a Q31 vector.
- *csi_min_q15* : Minimum value of a Q15 vector.
- *csi_min_q7* : Minimum value of a Q7 vector.

9.2.2 Description

Computes the minimum value of an array of data. The function returns both the minimum value and its position within the array. There are separate functions for floating-point, Q31, Q15, and Q7 data types.

9.2.3 Function Docmentation

9.2.3.1 csi_min_f32

```
void csi_min_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult, uint32_t␣
↪*pIndex)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: minimum value returned here.

`*pIndex`: index of minimum value returned here.

Returns:

none.

9.2.3.2 csi_min_q31

```
void csi_min_q31 (q31_t *pSrc, uint32_t blockSize, q31_t *pResult, uint32_t *pIndex)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: minimum value returned here.

`*pIndex`: index of minimum value returned here.

Returns:

none.

9.2.3.3 csi_min_q15

```
void csi_min_q15 (q15_t *pSrc, uint32_t blockSize, q15_t *pResult, uint16_t *pIndex)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: minimum value returned here.

`*pIndex`: index of minimum value returned here.

Returns:

none.

9.2.3.4 csi_min_q7

```
void csi_min_q7 (q7_t *pSrc, uint32_t blockSize, q7_t *pResult, uint16_t *pIndex)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: minimum value returned here.

`*pIndex`: index of minimum value returned here.

Returns:

none.

9.3 Mean

9.3.1 Functions

- *csi_mean_f32* : Mean value of a floating-point vector.
- *csi_mean_q31* : Mean value of a Q31 vector.
- *csi_mean_q15* : Mean value of a Q15 vector.
- *csi_mean_q7* : Mean value of a Q7 vector.

9.3.2 Description

Calculates the mean of the input vector. Mean is defined as the average of the elements in the vector. The underlying algorithm is used:

```
Result = (pSrc[0] + pSrc[1] + pSrc[2] + ... + pSrc[blockSize-1]) / blockSize;
```

There are separate functions for floating-point, Q31, Q15, and Q7 data types.

9.3.3 Function Docmentation

9.3.3.1 csi_mean_f32

```
void csi_mean_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vecto16

`*pResult`: mean value returned here.

Returns:

none.

9.3.3.2 csi_mean_q31

```
void csi_mean_q31 (q31_t *pSrc, uint32_t blockSize, q31_t *pResult)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: mean value returned here.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. The input is represented in 1.31 format and is accumulated in a 64-bit accumulator in 33.31 format. There is no risk of internal overflow with this approach, and the full precision of intermediate result is preserved. Finally, the accumulator is truncated to yield a result of 1.31 format.

9.3.3.3 csi_mean_q15

```
void csi_mean_q15 (q15_t *pSrc, uint32_t blockSize, q15_t *pResult)
```

Parameters:

`*pSrc`: points to the input vector.
`blockSize`: length of the input vector.
`*pResult`: mean value returned here.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using a 32-bit internal accumulator. The input is represented in 1.15 format and is accumulated in a 32-bit accumulator in 17.15 format. There is no risk of internal overflow with this approach, and the full precision of intermediate result is preserved. Finally, the accumulator is saturated and truncated to yield a result of 1.15 format.

9.3.3.4 csi_mean_q7

```
void csi_mean_q7 (q7_t *pSrc, uint32_t blockSize, q7_t *pResult)
```

Parameters:

`*pSrc`: points to the input vector.
`blockSize`: length of the input vector.
`*pResult`: mean value returned here.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using a 32-bit internal accumulator. The input is represented in 1.7 format and is accumulated in a 32-bit accumulator in 25.7 format. There is no risk of internal overflow with this approach, and the full precision of intermediate result is preserved. Finally, the accumulator is truncated to yield a result of 1.7 format.

9.4 Power

9.4.1 Functions

- *csi_power_int32* : Sum of the squares of the elements of a int32 vector.
- *csi_power_f32* : Sum of the squares of the elements of a floating-point vector.
- *csi_power_q31* : Sum of the squares of the elements of a Q31 vector.
- *csi_power_q15* : Sum of the squares of the elements of a Q15 vector.
- *csi_power_q7* : Sum of the squares of the elements of a Q7 vector.

9.4.2 Description

Calculates the sum of the squares of the elements in the input vector. The power_int32 is not support for E906/E907. The underlying algorithm is used:

```
Result = pSrc[0] * pSrc[0] + pSrc[1] * pSrc[1] + pSrc[2] * pSrc[2] + ... +␣
↪pSrc[blockSize-1] * pSrc[blockSize-1];
```

There are separate functions for floating point, Q31, Q15, and Q7 data types.

9.4.3 Function Docmentation

9.4.3.1 csi_power_int32

```
void csi_power_int32 (int32 *pSrc, uint32_t blockSize, q63_t *pResult)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: sum of the squares value returned here.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. The input is represented in 32-bit integer format. Intermediate multiplication yields 64-bit integer format, and this result is added with saturation to a 64-bit accumulator. And the result is between[0x0, 0x7fffffffffffff] in 64-bit integer format. So be careful to use it to avoid overflow.

9.4.3.2 csi_power_f32

```
void csi_power_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: sum of the squares value returned here.

Returns:

none.

9.4.3.3 csi_power_q31

```
void csi_power_q31 (q31_t *pSrc, uint32_t blockSize, q31_t *pResult)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: sum of the squares value returned here.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. The input is represented in 1.31 format. Intermediate multiplication yields a 2.62 format, and this result is truncated to 2.48 format by discarding the lower 14 bits. The 2.48 result is then added without saturation to a 64-bit accumulator in 16.48 format. With 15 guard bits in the accumulator, there is no risk of overflow, and the full precision of the intermediate multiplication is preserved. Finally, the return result is in 16.48 format.

9.4.3.4 csi_power_q15

```
void csi_power_q15 (q15_t *pSrc, uint32_t blockSize, q15_t *pResult)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: sum of the squares value returned here.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. The input is represented in 1.15 format. Intermediate multiplication yields a 2.30 format, and this result is added without saturation to a 64-bit accumulator in 34.30 format. With 33 guard bits in the accumulator, there is no risk of overflow, and the full precision of the intermediate multiplication is preserved. Finally, the return result is in 34.30 format.

9.4.3.5 csi_power_q7

```
void csi_power_q7 (q7_t *pSrc, uint32_t blockSize, q7_t *pResult)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: sum of the squares value returned here.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using a 32-bit internal accumulator. The input is represented in 1.7 format. Intermediate multiplication yields a 2.14 format, and this result is added without saturation to an accumulator in 18.14 format. With 17 guard bits in the accumulator, there is no risk of overflow, and the full precision of the intermediate multiplication is preserved. Finally, the return result is in 18.14 format.

9.5 Root mean square (RMS)

9.5.1 Functions

- *csi_rms_f32* : Root Mean Square of the elements of a floating-point vector.
- *csi_rms_q31* : Root Mean Square of the elements of a Q31 vector.
- *csi_rms_q15* : Root Mean Square of the elements of a Q15 vector.

9.5.2 Description

Calculates the Root Mean Sqaure of the elements in the input vector. The underlying algorithm is used:

```
Result = sqrt(((pSrc[0] * pSrc[0] + pSrc[1] * pSrc[1] + ... + pSrc[blockSize-
↪1] * pSrc[blockSize-1]) / blockSize));
```

There are separate functions for floating point, Q31, and Q15 data types.

9.5.3 Function Docmentation

9.5.3.1 csi_rms_f32

```
void csi_rms_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: sum of the squares value returned here.

Returns:

none.

9.5.3.2 csi_rms_q31

```
void csi_rms_q31 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: sum of the squares value returned here.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The input is represented in 1.31 format, and intermediate multiplication yields a 2.62 format. The accumulator maintains full precision of the intermediate multiplication results, but provides only a single guard bit. There is no saturation on intermediate additions. If the accumulator overflows, it wraps around and distorts the result. In order to avoid overflows completely, the input signal must be scaled down by log2(blockSize) bits, as a total of blockSize additions are performed internally. Finally, the 2.62 accumulator is right shifted by 31 bits to yield a 1.31 format value.

9.5.3.3 csi_rms_q15

```
void csi_rms_q15 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: sum of the squares value returned here.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. The input is represented in 1.15 format. Intermediate multiplication yields a 2.30 format, and this result is added without saturation to a 64-bit accumulator in 34.30 format. With 33 guard bits in the accumulator, there is no risk of overflow, and the full precision of the intermediate multiplication is preserved. Finally, the 34.30 result is truncated to 34.15 format by discarding the lower 15 bits, and then saturated to yield a result in 1.15 format.

9.6 Standard deviation

9.6.1 Functions

- *csi_std_f32* : Standard deviation of the elements of a floating-point vector.
- *csi_std_q31* : Standard deviation of the elements of a Q31 vector.
- *csi_std_q15* : Standard deviation of the elements of a Q15 vector.

9.6.2 Description

Calculates the standard deviation of the elements in the input vector. The underlying algorithm is used:

```
 Result = sqrt((sumOfSquares - sum2 / blockSize) / (blockSize - 1))

 where, sumOfSquares = pSrc[0] * pSrc[0] + pSrc[1] * pSrc[1] + ... +␣
↪pSrc[blockSize-1] * pSrc[blockSize-1]
sum = pSrc[0] + pSrc[1] + pSrc[2] + ... + pSrc[blockSize-1]
```

There are separate functions for floating point, Q31, and Q15 data types.

9.6.3 Function Docmentation

9.6.3.1 csi_std_f32

```
void csi_std_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: sum of the squares value returned here.

Returns:

none.

9.6.3.2 csi_std_q31

```
void csi_std_q31 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

Parameters:

*pSrc: points to the input vector.

blockSize: length of the input vector.

*pResult: sum of the squares value returned here.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The input is represented in 1.31 format, which is then downshifted by 8 bits which yields 1.23, and intermediate multiplication yields a 2.46 format. The accumulator maintains full precision of the intermediate multiplication results, but provides only a 16 guard bits. There is no saturation on intermediate additions. If the accumulator overflows it wraps around and distorts the result. In order to avoid overflows completely the input signal must be scaled down by log2(blockSize)-8 bits, as a total of blockSize additions are performed internally. After division, internal variables should be Q18.46 Finally, the 18.46 accumulator is right shifted by 15 bits to yield a 1.31 format value.

9.6.3.3 csi_std_q15

```
void csi_std_q15 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

Parameters:

*pSrc: points to the input vector.

blockSize: length of the input vector.

*pResult: sum of the squares value returned here.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. The input is represented in 1.15 format. Intermediate multiplication yields a 2.30 format, and this result is added without saturation to a 64-bit accumulator in 34.30 format. With 33 guard bits in the accumulator, there is no risk of overflow, and the full precision of the intermediate multiplication is preserved. Finally, the 34.30 result is truncated to 34.15 format by discarding the lower 15 bits, and then saturated to yield a result in 1.15 format.

9.7 Variance

9.7.1 Functions

- *csi_var_f32* : Variance of the elements of a floating-point vector.
- *csi_var_q31* : Variance of the elements of a Q31 vector.
- *csi_var_q15* : Variance of the elements of a Q15 vector.

9.7.2 Description

Calculates the variance of the elements in the input vector. The underlying algorithm is used:

```
 Result = (sumOfSquares - sum2 / blockSize) / (blockSize - 1)

 where, sumOfSquares = pSrc[0] * pSrc[0] + pSrc[1] * pSrc[1] + ... +␣
↪pSrc[blockSize-1] * pSrc[blockSize-1]
sum = pSrc[0] + pSrc[1] + pSrc[2] + ... + pSrc[blockSize-1]
```

There are separate functions for floating point, Q31, and Q15 data types.

9.7.3 Function Docmentation

9.7.3.1 csi_var_f32

```
void csi_var_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

Parameters:

`*pSrc`: points to the input vector.

`blockSize`: length of the input vector.

`*pResult`: sum of the squares value returned here.

Returns:

none.

9.7.3.2 csi_var_q31

```
void csi_var_q31 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

Parameters:

*pSrc: points to the input vector.

blockSize: length of the input vector.

*pResult: sum of the squares value returned here.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using an internal 64-bit accumulator. The input is represented in 1.31 format, which is then downshifted by 8 bits which yields 1.23, and intermediate multiplication yields a 2.46 format. The accumulator maintains full precision of the intermediate multiplication results, but provides only a 16 guard bits. There is no saturation on intermediate additions. If the accumulator overflows it wraps around and distorts the result. In order to avoid overflows completely the input signal must be scaled down by log2(blockSize)-8 bits, as a total of blockSize additions are performed internally. After division, internal variables should be Q18.46 Finally, the 18.46 accumulator is right shifted by 15 bits to yield a 1.31 format value.

9.7.3.3 csi_var_q15

```
void csi_var_q15 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

Parameters:

*pSrc: points to the input vector.

blockSize: length of the input vector.

*pResult: sum of the squares value returned here.

Returns:

none.

Scaling and Overflow Behavior:

The function is implemented using a 64-bit internal accumulator. The input is represented in 1.15 format. Intermediate multiplication yields a 2.30 format, and this result is added without saturation to a 64-bit accumulator in 34.30 format. With 33 guard bits in the accumulator, there is no risk of overflow, and the full precision of the intermediate multiplication is preserved. Finally, the 34.30 result is truncated to 34.15 format by discarding the lower 15 bits, and then saturated to yield a result in 1.15 format.

CHAPTER 10

Support Functions

10.1 Vector Copy

10.1.1 Functions

- *csi_copy_f32* : Copies the elements of a floating-point vector.
- *csi_copy_q31* : Copies the elements of a Q15 vector.
- *csi_copy_q15* : Copies the elements of a Q31 vector.
- *csi_copy_q7* : Copies the elements of a Q7 vector.

10.1.2 Description

Copies sample by sample from source vector to destination vector.

```
pDst[n] = pSrc[n];  0 <= n < blockSize.
```

There are separate functions for floating point, Q31, Q15, and Q7 data types.

10.1.3 Function Docmentation

10.1.3.1 csi_copy_f32

```
void csi_copy_f32 (float32_t *pSrc, float32_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

10.1.3.2 csi_copy_q31

```
void csi_copy_q31 (q31_t *pSrc, q31_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

10.1.3.3 csi_copy_q15

```
void csi_copy_q15 (q15_t *pSrc, q15_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

10.1.3.4 csi_copy_q7

```
void csi_copy_q7 (q7_t *pSrc, q7_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

10.2 Vector Fill

10.2.1 Functions

- *csi_fill_f32* : Fills a constant value into a floating-point vector.
- *csi_fill_q31* : Fills a constant value into a Q15 vector.
- *csi_fill_q15* : Fills a constant value into a Q31 vector.
- *csi_fill_q7* : Fills a constant value into a Q7 vector.

10.2.2 Description

Fills the destination vector with a constant value.

```
pDst[n] = value;   0 <= n < blockSize.
```

There are separate functions for floating point, Q31, Q15, and Q7 data types.

10.2.3 Function Docmentation

10.2.3.1 csi_fill_f32

```
void csi_fill_f32 (float32_t *pSrc, float32_t *pDst, uint32_t blockSize)
```

Parameters:

`value`: input value to be filled.

`*pDst`: points to output vector.

`blockSize`: length of the output vector.

Returns:

none.

10.2.3.2 csi_fill_q31

```
void csi_fill_q31 (q31_t *pSrc, q31_t *pDst, uint32_t blockSize)
```

Parameters:

`value`: input value to be filled.

`*pDst`: points to output vector.

`blockSize`: length of the output vector.

Returns:

none.

10.2.3.3 csi_fill_q15

```
void csi_fill_q15 (q15_t *pSrc, q15_t *pDst, uint32_t blockSize)
```

Parameters:

`value`: input value to be filled.

`*pDst`: points to output vector.

`blockSize`: length of the output vector.

Returns:

none.

10.2.3.4 csi_fill_q7

```
void csi_fill_q7 (q7_t *pSrc, q7_t *pDst, uint32_t blockSize)
```

Parameters:

`value`: input value to be filled.

`*pDst`: points to output vector.

`blockSize`: length of the output vector.

Returns:

none.

10.3 Convert 32-bit floating point value

10.3.1 Functions

- *csi_float_to_q15* : Converts the elements of the floating-point vector to Q15 vector.
- *csi_float_to_q31* : Converts the elements of the floating-point vector to Q31 vector.
- *csi_float_to_q7* : Converts the elements of the floating-point vector to Q7 vector.

10.3.2 Function Docmentation

10.3.2.1 csi_float_to_q15

```
void csi_float_to_q15 (float32_t *pSrc, q15_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

Description:

The equation used for the conversion process is:

```
pDst[n] = (q15_t)(pSrc[n] * 32768);   0 <= n < blockSize.
```

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] will be saturated.

10.3.2.2 csi_float_to_q31

```
void csi_float_to_q31 (float32_t *pSrc, q31_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

Description:

The equation used for the conversion process is:

```
pDst[n] = (q31_t)(pSrc[n] * 2147483648);   0 <= n < blockSize.
```

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q31 range[0x80000000 0x7FFFFFFF] will be saturated.

10.3.2.3 csi_float_to_q7

```
void csi_float_to_q7 (float32_t *pSrc, q7_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

Description:

The equation used for the conversion process is:

```
pDst[n] = (q7_t)(pSrc[n] * 128);   0 <= n < blockSize.
```

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q7 range [0x80 0x7F] will be saturated.

10.4 Convert 16-bit Integer value

10.4.1 Functions

- *csi_q15_to_float* : Converts the elements of the Q15 vector to floating-point vector.
- *csi_q15_to_q31* : Converts the elements of the Q15 vector to Q31 vector.
- *csi_q15_to_q7* : Converts the elements of the Q15 vector to Q7 vector.

10.4.2 Function Docmentation

10.4.2.1 csi_q15_to_float

```
void csi_q15_to_float (q15_t *pSrc, float32_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

Description:

The equation used for the conversion process is:

```
pDst[n] = (float32_t) pSrc[n] / 32768;   0 <= n < blockSize.
```

10.4.2.2 csi_q15_to_q31

```
void csi_q15_to_q31 (q15_t *pSrc, q31_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

Description:

The equation used for the conversion process is:

```
pDst[n] = (q31_t) pSrc[n] << 16;   0 <= n < blockSize.
```

10.4.2.3 csi_q15_to_q7

```
void csi_q15_to_q7 (float32_t *pSrc, q7_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

Description:

The equation used for the conversion process is:

```
pDst[n] = (q7_t) pSrc[n] >> 8;   0 <= n < blockSize.
```

10.5 Convert 32-bit Integer value

10.5.1 Functions

- *csi_q31_to_float* : Converts the elements of the Q31 vector to floating-point vector.
- *csi_q31_to_q15* : Converts the elements of the Q31 vector to Q15 vector.
- *csi_q31_to_q7* : Converts the elements of the Q31 vector to Q7 vector.

10.5.2 Function Docmentation

10.5.2.1 csi_q31_to_float

```
void csi_q31_to_float (q15_t *pSrc, float32_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

Description:

The equation used for the conversion process is:

```
pDst[n] = (float32_t) pSrc[n] / 2147483648;   0 <= n < blockSize.
```

10.5.2.2 csi_q31_to_q15

```
void csi_q31_to_q15 (q15_t *pSrc, q31_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

Description:

The equation used for the conversion process is:

```
pDst[n] = (q15_t) pSrc[n] >> 16;   0 <= n < blockSize.
```

10.5.2.3 csi_q31_to_q7

```
void csi_q31_to_q7 (float32_t *pSrc, q7_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

Description:

The equation used for the conversion process is:

```
pDst[n] = (q7_t) pSrc[n] >> 24;   0 <= n < blockSize.
```

10.6 Convert 32-bit Integer value

10.6.1 Functions

- *csi_q7_to_float* : Converts the elements of the Q7 vector to floating-point vector.
- *csi_q7_to_q15* : Converts the elements of the Q7 vector to Q15 vector.
- *csi_q7_to_q31* : Converts the elements of the Q7 vector to Q31 vector.

10.6.2 Function Docmentation

10.6.2.1 csi_q7_to_float

```
void csi_q7_to_float (q15_t *pSrc, float32_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

Description:

The equation used for the conversion process is:

```
pDst[n] = (float32_t) pSrc[n] / 128;   0 <= n < blockSize.
```

10.6.2.2 csi_q7_to_q15

```
void csi_q7_to_q15 (q15_t *pSrc, q31_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

Description:

The equation used for the conversion process is:

```
pDst[n] = (q15_t) pSrc[n] << 8;   0 <= n < blockSize.
```

10.6.2.3 csi_q7_to_q31

```
void csi_q7_to_q31 (float32_t *pSrc, q7_t *pDst, uint32_t blockSize)
```

Parameters:

`*pSrc`: points to input vector.

`*pDst`: points to output vector.

`blockSize`: length of the input vector.

Returns:

none.

Description:

The equation used for the conversion process is:

```
pDst[n] = (q31_t) pSrc[n] << 24;   0 <= n < blockSize.
```

CHAPTER 11

Transform Functions

11.1 Complex FFT Functions

11.1.1 Functions

- *csi_cfft_f32* : Processing function for the floating-point complex FFT.
- *csi_cfft_q15* : Processing function for the Q15 complex FFT.
- *csi_cfft_q31* : Processing function for the Q31 complex FFT.

11.1.2 Description

The Fast Fourier Transform (FFT) is an efficient algorithm for computing the Discrete Fourier Transform (DFT). The FFT can be orders of magnitude faster than the DFT, especially for long lengths. The algorithms described in this section operate on complex data. A separate set of functions is devoted to handling of real sequences.

There are separate algorithms for handling floating-point, Q15, and Q31 data types. The algorithms available for each data type are described next.

The FFT functions operate in-place. That is, the array holding the input data will also be used to hold the corresponding result. The input data is complex and contains 2*fftLen interleaved values as shown below.

```
{real[0], imag[0], real[1], imag[1],..}
```

The FFT result will be contained in the same array and the frequency domain values will have the same interleaving.

Floating-point

The floating-point complex FFT uses a mixed-radix algorithm. Multiple radix-8 stages are performed along with a single radix-2 or radix-4 stage, as needed. The algorithm supports lengths of [16, 32, 64, ···, 4096] and each length uses a different twiddle factor table.

The function uses the standard FFT definition and output values may grow by a factor of fftLen when computing the forward transform. The inverse transform includes a scale of 1/fftLen as part of the calculation and this matches the textbook definition of the inverse FFT.

Pre-initialized data structures containing twiddle factors and bit reversal tables are provided and defined in csi_const_structs.h. Include this header in your function and then pass one of the constant structures as an argument to csi_cfft_f32. For example:

```
csi_cfft_f32(csi_cfft_sR_f32_len64, pSrc, 1, 1)
```

computes a 64-point inverse complex FFT including bit reversal. The data structures are treated as constant data and not modified during the calculation. The same data structure can be reused for multiple transforms including mixing forward and inverse transforms.

Earlier releases of the library provided separate radix-2 and radix-4 algorithms that operated on floating-point data. These functions are still provided but are deprecated. The older functions are slower and less general than the new functions.

An example of initialization of the constants for the csi_cfft_f32 function follows:

```
const static csi_cfft_instance_f32 *S;
...
  switch (length) {
  case 16:
    S = &csi_cfft_sR_f32_len16;
    break;
  case 32:
    S = &csi_cfft_sR_f32_len32;
    break;
  case 64:
    S = &csi_cfft_sR_f32_len64;
    break;
  case 128:
    S = &csi_cfft_sR_f32_len128;
    break;
  case 256:
    S = &csi_cfft_sR_f32_len256;
    break;
  case 512:
    S = &csi_cfft_sR_f32_len512;
    break;
  case 1024:
    S = &csi_cfft_sR_f32_len1024;
    break;
  case 2048:
    S = &csi_cfft_sR_f32_len2048;
```

(continues on next page)

(continued from previous page)

```
    break;
  case 4096:
    S = &csi_cfft_sR_f32_len4096;
    break;
  }
```

Q15 and Q31

The floating-point complex FFT uses a mixed-radix algorithm. Multiple radix-4 stages are performed along with a single radix-2 stage, as needed. The algorithm supports lengths of [16, 32, 64, ⋯, 4096] and each length uses a different twiddle factor table.

The function uses the standard FFT definition and output values may grow by a factor of fftLen when computing the forward transform. The inverse transform includes a scale of 1/fftLen as part of the calculation and this matches the textbook definition of the inverse FFT.

Pre-initialized data structures containing twiddle factors and bit reversal tables are provided and defined in csi_const_structs.h. Include this header in your function and then pass one of the constant structures as an argument to csi_cfft_q31. For example:

```
csi_cfft_q31(csi_cfft_sR_q31_len64, pSrc, 1, 1)
```

computes a 64-point inverse complex FFT including bit reversal. The data structures are treated as constant data and not modified during the calculation. The same data structure can be reused for multiple transforms including mixing forward and inverse transforms.

Earlier releases of the library provided separate radix-2 and radix-4 algorithms that operated on floating-point data. These functions are still provided but are deprecated. The older functions are slower and less general than the new functions.

An example of initialization of the constants for the csi_cfft_q31 function follows:

```
const static csi_cfft_instance_q31 *S;
...
  switch (length) {
  case 16:
    S = &csi_cfft_sR_q31_len16;
    break;
  case 32:
    S = &csi_cfft_sR_q31_len32;
    break;
  case 64:
    S = &csi_cfft_sR_q31_len64;
    break;
  case 128:
    S = &csi_cfft_sR_q31_len128;
    break;
  case 256:
    S = &csi_cfft_sR_q31_len256;
```

(continues on next page)

(continued from previous page)

```
  break;
case 512:
  S = &csi_cfft_sR_q31_len512;
  break;
case 1024:
  S = &csi_cfft_sR_q31_len1024;
  break;
case 2048:
  S = &csi_cfft_sR_q31_len2048;
  break;
case 4096:
  S = &csi_cfft_sR_q31_len4096;
  break;
}
```

11.1.3 Function Docmentation

11.1.3.1 csi_cfft_f32

```
void csi_cfft_f32 (const csi_cfft_instance_f32 *S, float32_t *p1, uint8_t ifftFlag,␣
↪uint8_t bitReverseFlag)
```

Parameters:

`*S`: points to an instance of the CFFT structure.

`*p1`: points to the complex data buffer of size 2*fftLen. Processing occurs in-place.

`ifftFlag`: flag that selects forward (ifftFlag=0) or inverse (ifftFlag=1) transform.

`bitReverseFlag`: flag that enables (bitReverseFlag=1) or disables (bitReverseFlag=0) bit reversal of output.

Returns:

none.

11.1.3.2 csi_cfft_q15

```
void csi_cfft_q15 (const csi_cfft_instance_q15 *S, q15_t *p1, uint8_t ifftFlag, uint8_
↪t bitReverseFlag)
```

Parameters:

`*S`: points to an instance of the CFFT structure.

`*p1`: points to the complex data buffer of size 2*fftLen. Processing occurs in-place.

`ifftFlag`: flag that selects forward (ifftFlag=0) or inverse (ifftFlag=1) transform.

`bitReverseFlag`: flag that enables (bitReverseFlag=1) or disables (bitReverseFlag=0) bit reversal of output.

Returns:

none.

11.1.3.3 csi_cfft_q31

```
void csi_cfft_q31 (const csi_cfft_instance_q31 *S, q31_t *p1, uint8_t ifftFlag, uint8_
↪t bitReverseFlag)
```

Parameters:

`*S`: points to an instance of the CFFT structure.

`*p1`: points to the complex data buffer of size 2*fftLen. Processing occurs in-place.

`ifftFlag`: flag that selects forward (ifftFlag=0) or inverse (ifftFlag=1) transform.

`bitReverseFlag`: flag that enables (bitReverseFlag=1) or disables (bitReverseFlag=0) bit reversal of output.

Returns:

none.

11.2 Real FFT Functions

11.2.1 Functions

- *csi_rfft_fast_f32* : Processing function for the floating-point real FFT/IFFT.
- *csi_rfft_q15* : Processing function for the Q15 real FFT/IFFT.
- *csi_rfft_q31* : Processing function for the Q31 real FFT/IFFT.
- *csi_rfft_init_f32* : Initialization function for the floating-point real FFT/IFFT.
- *csi_rfft_init_q15* : Initialization function for the Q15 real FFT/IFFT.
- *csi_rfft_init_q31* : Initialization function for the Q31 real FFT/IFFT.

11.2.2 Description

The CSI DSP library includes specialized algorithms for computing the FFT of real data sequences. The FFT is defined over complex data but in many applications the input is real. Real FFT algorithms take advantage of the symmetry properties of the FFT and have a speed advantage over complex algorithms of the same length.

The Fast RFFT algorith relays on the mixed radix CFFT that save processor usage.

The real length N forward FFT of a sequence is computed using the steps shown below.

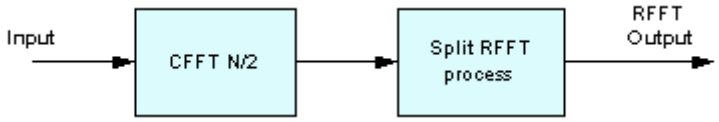


Figure11.1: Real Fast Fourier Transform

The real sequence is initially treated as if it were complex to perform a CFFT. Later, a processing stage reshapes the data to obtain half of the frequency spectrum in complex format. Except the first complex number that contains the two real numbers X[0] and X[N/2] all the data is complex. In other words, the first complex sample contains two real values packed.

The input for the inverse RFFT should keep the same format as the output of the forward RFFT. A first processing stage pre-process the data to later perform an inverse CFFT.

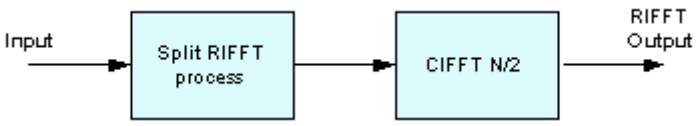


Figure11.2: Real Inverse Fast Fourier Transform

The algorithms for floating-point, Q15, and Q31 data are slightly different and we describe each algorithm in turn.

Floating-point

The main functions are csi_rfft_fast_f32() and csi_rfft_fast_init_f32(). The older functions csi_rfft_f32() and csi_rfft_init_f32() have been deprecated but are still documented.

The FFT of a real N-point sequence has even symmetry in the frequency domain. The second half of the data equals the conjugate of the first half flipped in frequency:

```
*X[0] - real data
*X[1] - complex data
*X[2] - complex data
*...
*X[fftLen/2-1] - complex data
*X[fftLen/2] - real data
*X[fftLen/2+1] - conjugate of X[fftLen/2-1]
*X[fftLen/2+2] - conjugate of X[fftLen/2-2]
*...
*X[fftLen-1] - conjugate of X[1]
```

Looking at the data, we see that we can uniquely represent the FFT using only

```
*N/2+1 samples:
*X[0] - real data
*X[1] - complex data
*X[2] - complex data
*...
*X[fftLen/2-1] - complex data
*X[fftLen/2] - real data
```

Looking more closely we see that the first and last samples are real valued. So their image part can be padding with zero. and we can thus represent the FFT of an N-point real sequence by (N/2 + 1) complex values:

```
*X[0] - complex data, its image part is zero.
*X[1] - complex data
*X[2] - complex data
*...
*X[fftLen/2-1] - complex data
*X[fftLen/2] - complex data, its image part is zero.
```

The real FFT functions pack the frequency domain data in this fashion. The forward transform outputs the data in this form and the inverse transform expects input data in this form. The function always performs the needed bitreversal so that the input and output data is always in normal order. The functions support lengths of [32, 64, 128, ⋯, 4096, 8192] samples.

The forward and inverse real FFT functions apply the standard FFT scaling; no scaling on the forward transform and 1/fftLen scaling on the inverse transform.

Q15 and Q31

The real algorithms are defined in a similar manner and utilize (N/2+1) complex transforms behind the scenes.

The complex transforms used internally include scaling to prevent fixed-point overflows. The overall scaling equals 1/(fftLen/2).

A separate instance structure must be defined for each transform used but twiddle factor and bit reversal tables can be reused.

How to use the functions

There are two ways to use these functions. The recommended one is to call them straightforwardly without initialization, which has a adventage of less memory usage. And it is more convenient to use RFFT functions, when the initialization function is not needed.

The other way needs an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Initializes twiddle factor table and bit reversal table pointers.
- Initializes the internal complex FFT data structure.

For these two ways, they both need to initialize the related instance structure. The difference between them is that, the recommended way initializes the structures manually and the second one initializes the structures by the associated initialization function. Alougth, these two ways are both available, but the one without initialization is perfered. The initialized structures are as follows:

```
*csi_rfft_instance_q31 S = {fftLenReal, fftLenBy2, ifftFlagR,␣
↪bitReverseFlagR, twidCoefRModifier, pTwiddleAReal, pCfft};
*csi_rfft_instance_q15 S = {fftLenReal, fftLenBy2, ifftFlagR,␣
↪bitReverseFlagR, twidCoefRModifier, pTwiddleAReal, pCfft};
```

where fftLenReal is the length of the real transform; fftLenBy2 length of the internal complex transform. ifftFlagR Selects forward (=0) or inverse (=1) transform. bitReverseFlagR Selects bit reversed output (=0) or normal order output (=1). twidCoefRModifier stride modifier for the twiddle factor table. The value is based on the FFT length; pTwiddleARealpoints to the A array of twiddle coefficients; pCfft points to the CFFT Instance structure. The CFFT structure must also be initialized. Refer to csi_cfft_radix4_f32() for details regarding static initialization of the complex FFT instance structure.

11.2.3 Function Docmentation

11.2.3.1 csi_rfft_fast_f32

```
void csi_rfft_fast_f32 (csi_rfft_fast_instance_f32 *S, float32_t *p, float32_t *pOut,␣
↪uint8_t ifftFlag)
```

Parameters:

`*S`: points to an instance of the RFFT structure.

`*p`: points to the input buffer.

`*pOut`: points to the output buffer.

`ifftFlag`: RFFT if flag is 0, RIFFT if flag is 1

Returns:

none.

Recommended Usage

For each length of input, the usage is as follows:

```
csi_rfft_fast_f32(&csi_rfft_sR_f32_len32, pSrc, pDst, ifftFlag);
csi_rfft_fast_f32(&csi_rfft_sR_f32_len64, pSrc, pDst, ifftFlag);
....
csi_rfft_fast_f32(&csi_rfft_sR_f32_len4096, pSrc, pDst, ifftFlag);
csi_rfft_fast_f32(&csi_rfft_sR_f32_len8192, pSrc, pDst, ifftFlag);
```

11.2.3.2 csi_rfft_q15

```
void csi_rfft_q15 (const csi_rfft_instance_q15 *S, q15_t *pSrc, q15_t *pDst)
```

Parameters:

`*S`: points to an instance of the RFFT structure.

`*p`: points to the input buffer.

`*pOut`: points to the output buffer.

Returns:

none.

Recommended Usage

For each length of input, the usage is as follows:

```
csi_rfft_q15(&csi_rfft_sR_q15_len32, pSrc, pDst);
csi_rfft_q15(&csi_rfft_sR_q15_len64, pSrc, pDst);
....
csi_rfft_q15(&csi_rfft_sR_q15_len4096, pSrc, pDst);
csi_rfft_q15(&csi_rfft_sR_q15_len8192, pSrc, pDst);
```

Input an output formats:

Internally input is downscaled by 2 for every stage to avoid saturations inside CFFT/CIFFT process. Hence the output format is different for different RFFT sizes. The input and output formats for different RFFT sizes and number of bits to upscale are mentioned in the tables below for RFFT and RIFFT:

RFFT Size	Input Format	Output Format	Number of bits to upscale
32	1.15	5.11	4
64	1.15	6.10	5
128	1.15	7.9	6
256	1.15	8.8	7
512	1.15	9.7	8
1024	1.15	10.6	9
2048	1.15	11.5	10
4096	1.15	12.4	11
8192	1.15	13.3	12

RIFFT Size	Input Format	Output Format	Number of bits to upscale
32	1.15	5.11	0
64	1.15	6.10	0
128	1.15	7.9	0
256	1.15	8.8	0
512	1.15	9.7	0
1024	1.15	10.6	0
2048	1.15	11.5	0
4096	1.15	12.4	0
8192	1.15	13.3	0

11.2.3.3 csi_rfft_q31

```
void csi_rfft_q31 (const csi_rfft_instance_q31 *S, q31_t *pSrc, q31_t *pDst)
```

Parameters:

`*S`: points to an instance of the RFFT structure.

`*p`: points to the input buffer.

`*pOut`: points to the output buffer.

Returns:

none.

Recommended Usage

For each length of input, the usage is as follows:

```
csi_rfft_q31(&csi_rfft_sR_q31_len32, pSrc, pDst);
csi_rfft_q31(&csi_rfft_sR_q31_len64, pSrc, pDst);
....
csi_rfft_q31(&csi_rfft_sR_q31_len4096, pSrc, pDst);
csi_rfft_q31(&csi_rfft_sR_q31_len8192, pSrc, pDst);
```

Input an output formats:

Internally input is downscaled by 2 for every stage to avoid saturations inside CFFT/CIFFT process. Hence the output format is different for different RFFT sizes. The input and output formats for different RFFT sizes and number of bits to upscale are mentioned in the tables below for RFFT and RIFFT:

RFFT Size	Input Format	Output Format	Number of bits to upscale
32	1.31	5.27	4
64	1.31	6.26	5
128	1.31	7.25	6
256	1.31	8.24	7
512	1.31	9.23	8
1024	1.31	10.22	9
2048	1.31	11.21	10
4096	1.31	12.20	11
8192	1.31	13.19	12

RIFFT Size	Input Format	Output Format	Number of bits to upscale
32	1.31	5.27	0
64	1.31	6.26	0
128	1.31	7.25	0
256	1.31	8.24	0
512	1.31	9.23	0
1024	1.31	10.22	0
2048	1.31	11.21	0
4096	1.31	12.20	0
8192	1.31	13.19	0

11.2.3.4 csi_rfft_init_f32

```
csi_status csi_rfft_init_f32 (csi_rfft_instance_f32 *S, csi_cfft_radix4_instance_f32␣
↪*S_CFFT, uint32_t fftLenReal, uint32_t ifftFlagR, uint32_t bitReverseFlag)
```

Parameters:

`*S`: points to an instance of the RFFT structure.

`*S_CFFT`: points to an instance of the CFFT structure.

`fftLenReal`: length of the FFT.

`ifftFlagR`: flag that selects forward (ifftFlagR=0) or inverse (ifftFlagR=1) transform.

`bitReverseFlag`: flag that enables (bitReverseFlag=1) or disables (bitReverseFlag=0) bit reversal of output.

Returns:

The function returns CSKY_MATH_SUCCESS if initialization is successful or CSKY_MATH_ARGUMENT_ERROR if fftLenReal is not a supported value.

Description:

The parameter fftLenReal Specifies length of RFFT/RIFFT Process. Supported FFT Lengths are 128, 512, 2048.

The parameter ifftFlagR controls whether a forward or inverse transform is computed. Set(=1) ifftFlagR to calculate RIFFT, otherwise RFFT is calculated.

The parameter bitReverseFlag controls whether output is in normal order or bit reversed order. Set(=1) bitReverseFlag for output to be in normal order otherwise output is in bit reversed order.

This function also initializes Twiddle factor table.

11.2.3.5 csi_rfft_init_q15

```
csi_status csi_rfft_init_q15 (csi_rfft_instance_q15 *S, uint32_t fftLenReal, uint32_t␣
↪ifftFlagR, uint32_t bitReverseFlag)
```

Parameters:

`*S`: points to an instance of the RFFT structure.

`*S_CFFT`: points to an instance of the CFFT structure.

`fftLenReal`: length of the FFT.

`ifftFlagR`: flag that selects forward (ifftFlagR=0) or inverse (ifftFlagR=1) transform.

`bitReverseFlag`: flag that enables (bitReverseFlag=1) or disables (bitReverseFlag=0) bit reversal of output.

Returns:

The function returns CSKY_MATH_SUCCESS if initialization is successful or CSKY_MATH_ARGUMENT_ERROR if fftLenReal is not a supported value.

Description:

The parameter fftLenReal Specifies length of RFFT/RIFFT Process. Supported FFT Lengths are 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192.

The parameter ifftFlagR controls whether a forward or inverse transform is computed. Set(=1) ifftFlagR to calculate RIFFT, otherwise RFFT is calculated.

The parameter bitReverseFlag controls whether output is in normal order or bit reversed order. Set(=1) bitReverseFlag for output to be in normal order otherwise output is in bit reversed order.

This function also initializes Twiddle factor table.

Note

This initialization function is recommended not to used, the function *csi_rfft_q15()* can be used straightforwardly by the manully initialized instance instructions for each length.

11.2.3.6 csi_rfft_init_q31

```
csi_status csi_rfft_init_q31 (csi_rfft_instance_q31 *S, uint32_t fftLenReal, uint32_t␣
↪ifftFlagR, uint32_t bitReverseFlag)
```

Parameters:

`*S`: points to an instance of the RFFT structure.

`*S_CFFT`: points to an instance of the CFFT structure.

`fftLenReal`: length of the FFT.

`ifftFlagR`: flag that selects forward (ifftFlagR=0) or inverse (ifftFlagR=1) transform.

`bitReverseFlag`: flag that enables (bitReverseFlag=1) or disables (bitReverseFlag=0) bit reversal of output.

Returns:

The function returns CSKY_MATH_SUCCESS if initialization is successful or CSKY_MATH_ARGUMENT_ERROR if fftLenReal is not a supported value.

Description:

The parameter fftLenReal Specifies length of RFFT/RIFFT Process. Supported FFT Lengths are 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192.

The parameter ifftFlagR controls whether a forward or inverse transform is computed. Set(=1) ifftFlagR to calculate RIFFT, otherwise RFFT is calculated.

The parameter bitReverseFlag controls whether output is in normal order or bit reversed order. Set(=1) bitReverseFlag for output to be in normal order otherwise output is in bit reversed order.

This function also initializes Twiddle factor table.

Note

This initialization function is recommended not to used, the function *csi_rfft_q31()* can be used straightforwardly by the manully initialized instance instructions for each length.

11.3 DCT Type IV Functions

11.3.1 Functions

- *csi_dct4_f32* : Processing function for the floating-point DCT4/IDCT4.
- *csi_dct4_q15* : Processing function for the Q15 real DCT4/IDCT4.
- *csi_dct4_q31* : Processing function for the Q31 real DCT4/IDCT4.
- *csi_dct4_init_f32* : Initialization function for the floating-point real DCT4/IDCT4.
- *csi_dct4_init_q15* : Initialization function for the Q15 real DCT4/IDCT4.
- *csi_dct4_init_q31* : Initialization function for the Q31 real DCT4/IDCT4.

11.3.2 Description

Representation of signals by minimum number of values is important for storage and transmission. The possibility of large discontinuity between the beginning and end of a period of a signal in DFT can be avoided by extending the signal so that it is even-symmetric. Discrete Cosine Transform (DCT) is constructed such that its energy is heavily concentrated in the lower part of the spectrum and is very widely used in signal and image coding applications. The family of DCTs (DCT type- 1,2,3,4) is the outcome of different combinations of homogeneous boundary conditions. DCT has an excellent energy-packing capability, hence has many applications and in data compression in particular.

DCT is essentially the Discrete Fourier Transform(DFT) of an even-extended real signal. Reordering of the input data makes the computation of DCT just a problem of computing the DFT of a real signal with a few additional operations. This approach provides regular, simple, and very efficient DCT algorithms for practical hardware and software implementations.

DCT type-II can be implemented using Fast fourier transform (FFT) internally, as the transform is applied on real values, Real FFT can be used. DCT4 is implemented using DCT2 as their implementations are similar except with some added pre-processing and post-processing. DCT2 implementation can be described in the following steps:

- Re-ordering input
- Calculating Real FFT
- Multiplication of weights and Real FFT output and getting real part from the product.

This process is explained by the block diagram below:

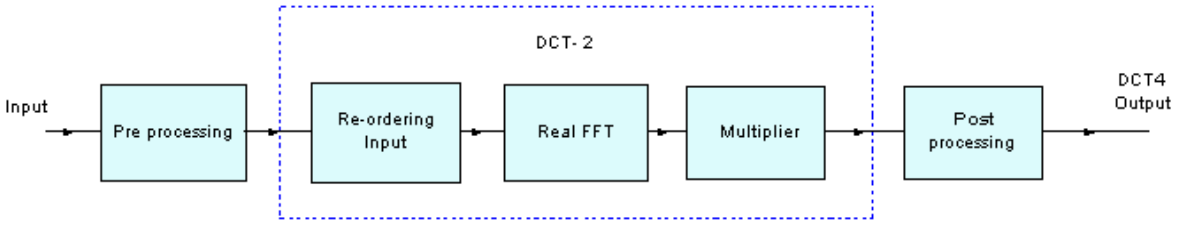


Figure11.3: Discrete Cosine Transform - type-IV

Algorithm:

$$X_c(k) = \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} x(n) \cos\left[\left(n + \frac{1}{2}\right)\left(k + \frac{1}{2}\right)\frac{\pi}{N}\right]$$

The N-point type-IV DCT is defined as a real, linear transformation by the formula:

where k = 0,1,2,⋯..N-1

Its inverse is defined as follows:

$$x(n) = \sqrt{\frac{2}{N}} \sum_{k=0}^{N-1} X_c(k) \cos\left[\left(n + \frac{1}{2}\right)\left(k + \frac{1}{2}\right)\frac{\pi}{N}\right]$$

where n = 0,1,2,⋯..N-1

The DCT4 matrices become involutory (i.e. they are self-inverse) by multiplying with an overall scale factor of sqrt(2/N). The symmetry of the transform matrix indicates that the fast algorithms for the forward and inverse transform computation are identical. Note that the implementation of Inverse DCT4 and DCT4 is same, hence same process function can be used for both.

Lengths supported by the transform:

As DCT4 internally uses Real FFT, it supports all the lengths supported by csi_rfft_fast_f32(). The library provides separate functions for Q15, Q31, and floating-point data types.

Instance Structure

The instances for Real FFT and FFT, cosine values table and twiddle factor table are stored in an instance data structure. A separate instance structure must be defined for each transform. There are separate instance structure declarations for each of the 3 supported data types.

How to use the function

Same as the RFFT functions, the DCT4 functions also have two ways to use them. The recommended one is to call them straightforwardly without initialization, which has a adventage of less memory usage. And it is more convenient to use RFFT functions, when the initialization function is not needed.

The other way needs an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Initializes Real FFT as its process function is used internally in DCT4, by calling csi_rfft_fast_init_f32().

For these two ways, they both need to initialize the related instance structure. The difference between them is that, the recommended way initializes the structures manually and the second one initializes the structures by the associated initialization function. Alougth, these two ways are both available, but the one without initialization is perfered. The initialized instance structures are as follows:

```
*csi_dct4_instance_f32 S = {N, Nby2, normalize, pTwiddle, pCosFactor, pRfft,␣
↪pCfft};
*csi_dct4_instance_q31 S = {N, Nby2, normalize, pTwiddle, pCosFactor, pRfft,␣
↪pCfft};
*csi_dct4_instance_q15 S = {N, Nby2, normalize, pTwiddle, pCosFactor, pRfft,␣
↪pCfft};
```

where N is the length of the DCT4; Nby2 is half of the length of the DCT4; normalize is normalizing factor used and is equal to sqrt(2/N); pTwiddle points to the twiddle factor table; pCosFactor points to the cosFactor table; pRfft points to the real FFT instance; pCfft points to the complex FFT instance; The CFFT and RFFT structures also needs to be initialized, refer to csi_cfft_f32() and csi_rfft_fast_f32() respectively for details regarding static initialization.

Fixed-Point Behavior

Care must be taken when using the fixed-point versions of the DCT4 transform functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

11.3.3 Function Docmentation

11.3.3.1 csi_dct4_f32

```
void csi_dct4_f32 (const csi_dct4_instance_f32 *S, float32_t *pState, float32_t␣
↪*pInlineBuffer)
```

Parameters:

`*S`: points to an instance of the DCT4/IDCT4 structure.

`*pState`: points to the state buffer.

`*pInlineBuffer`: points to the in-place input and output buffer.

Returns:

none.

Recommended Usage

For each length of input, the usage is as follows:

```
csi_dct4_f32(&csi_dct4_sR_f32_len128, pState, pSrc);
csi_dct4_f32(&csi_dct4_sR_f32_len512, pState, pSrc);
csi_dct4_f32(&csi_dct4_sR_f32_len2048, pState, pSrc);
csi_dct4_f32(&csi_dct4_sR_f32_len8192, pState, pSrc);
```

11.3.3.2 csi_dct4_q15

```
void csi_dct4_q15 (const csi_dct4_instance_q15 *S, q15_t *pState, q15_t␣
↪*pInlineBuffer)
```

Parameters:

`*S`: points to an instance of the DCT4/IDCT4 structure.

`*pState`: points to the state buffer.

`*pInlineBuffer`: points to the in-place input and output buffer.

Returns:

none.

Recommended Usage

For each length of input, the usage is as follows:

```
csi_dct4_q15(&csi_dct4_sR_q15_len128, pState, pSrc);
csi_dct4_q15(&csi_dct4_sR_q15_len512, pState, pSrc);
csi_dct4_q15(&csi_dct4_sR_q15_len2048, pState, pSrc);
csi_dct4_q15(&csi_dct4_sR_q15_len8192, pState, pSrc);
```

Input an output formats:

Input samples need to be downscaled by 1 bit to avoid saturations in the Q15 DCT process, Internally inputs are downscaled in the RFFT process function to avoid overflows. Number of bits downscaled, depends on the size of the transform. The input and output formats for different DCT sizes and number of bits to upscale are mentioned in the table below:

DCT Size	input format	Output format	Number of bits to upscale
8192	1.15	13.3	12
2048	1.15	11.5	10
512	1.15	9.7	8
128	1.15	7.9	6

11.3.3.3 csi_dct4_q31

```
void csi_dct4_q31 (const csi_dct4_instance_q31 *S, q31_t *pState, q31_t␣
↪*pInlineBuffer)
```

Parameters:

`*S`: points to an instance of the DCT4/IDCT4 structure.

`*pState`: points to the state buffer.

`*pInlineBuffer`: points to the in-place input and output buffer.

Returns:

none.

Recommended Usage

For each length of input, the usage is as follows:

```
csi_dct4_q31(&csi_dct4_sR_q31_len128, pState, pSrc);
csi_dct4_q31(&csi_dct4_sR_q31_len512, pState, pSrc);
csi_dct4_q31(&csi_dct4_sR_q31_len2048, pState, pSrc);
csi_dct4_q31(&csi_dct4_sR_q31_len8192, pState, pSrc);
```

Input an output formats:

Input samples need to be downscaled by 1 bit to avoid saturations in the Q31 DCT process, as the conversion from DCT2 to DCT4 involves one subtraction. Internally inputs are downscaled in the RFFT process function to avoid overflows. Number of bits downscaled, depends on the size of the transform. The input and output formats for different DCT sizes and number of bits to upscale are mentioned in the table below:

DCT Size	input format	Output format	Number of bits to upscale
8192	2.30	14.18	12
2048	2.30	12.20	11
512	2.30	10.22	9
128	2.30	8.24	7

11.3.3.4 csi_dct4_init_f32

```
csi_status csi_dct4_init_f32 (csi_dct4_instance_f32 *S, csi_rfft_fast_instance_f32 *S_
↪RFFT, csi_cfft_radix4_instance_f32 *S_CFFT, uint16_t N, uint16_t Nby2, float32_t␣
↪normalize)
```

Parameters:

`*S`: points to an instance of the DCT4/IDCT4 structure.

`*S_RFFT`: points to an instance of the RFFT/RIFFT structure.

`*S_CFFT`: points to an instance of the CFFT/CIFFT structure.

`N`: length of the DCT4.

`Nby2`: half of the length of the DCT4.

`normalize`: normalizing factor.

Returns:

csi_status function returns CSKY_MATH_SUCCESS if initialization is successful or CSKY_MATH_ARGUMENT_ERROR if fftLenReal is not a supported transform length.

Normalizing factor:

The normalizing factor is sqrt(2/N), which depends on the size of transform N. Floating-point normalizing factors are mentioned in the table below for different DCT sizes:

DCT Size	Normalizing factor value
8192	0.015625
2048	0.03125
512	0.0625
128	0.125

Note

This initialization function is recommended not to used, the function *csi_dct4_f32()* can be used straightforwardly by the manully initialized instance instructions for each length.

11.3.3.5 csi_dct4_init_q15

```
csi_status csi_dct4_init_q15 (csi_dct4_instance_q15 *S, csi_rfft_instance_q15 *S_RFFT,
↪ csi_cfft_radix4_instance_q15 *S_CFFT, uint16_t N, uint16_t Nby2, q15_t normalize)
```

Parameters:

`*S`: points to an instance of the DCT4/IDCT4 structure.

`*S_RFFT`: points to an instance of the RFFT/RIFFT structure.

`*S_CFFT`: points to an instance of the CFFT/CIFFT structure.

`N`: length of the DCT4.

`Nby2`: half of the length of the DCT4.

`normalize`: normalizing factor.

Returns:

csi_status function returns CSKY_MATH_SUCCESS if initialization is successful or CSKY_MATH_ARGUMENT_ERROR if fftLenReal is not a supported transform length.

Normalizing factor:

The normalizing factor is sqrt(2/N), which depends on the size of transform N. Normalizing factors in 1.15 format are mentioned in the table below for different DCT sizes:

DCT Size	Normalizing factor value (hexadecimal)
8192	0x200
2048	0x400
512	0x800
128	0x1000

Note

This initialization function is recommended not to used, the function *csi_dct4_q15()* can be used straightforwardly by the manully initialized instance instructions for each length.

11.3.3.6 csi_dct4_init_q31

```
csi_status csi_dct4_init_q31 (csi_dct4_instance_q31 *S, csi_rfft_instance_q31 *S_RFFT,
↪ csi_cfft_radix4_instance_q31 *S_CFFT, uint16_t N, uint16_t Nby2, q31_t normalize)
```

Parameters:

`*S`: points to an instance of the DCT4/IDCT4 structure.

`*S_RFFT`: points to an instance of the RFFT/RIFFT structure.

`*S_CFFT`: points to an instance of the CFFT/CIFFT structure.

`N`: length of the DCT4.

`Nby2`: half of the length of the DCT4.

`normalize`: normalizing factor.

Returns:

csi_status function returns CSKY_MATH_SUCCESS if initialization is successful or CSKY_MATH_ARGUMENT_ERROR if fftLenReal is not a supported transform length.

Normalizing factor:

The normalizing factor is sqrt(2/N), which depends on the size of transform N. Normalizing factors in 1.31 format are mentioned in the table below for different DCT sizes:

DCT Size	Normalizing factor value (hexadecimal)
8192	0x2000000
2048	0x4000000
512	0x8000000
128	0x10000000

Note

This initialization function is recommended not to used, the function *csi_dct4_q31()* can be used straightforwardly by the manully initialized instance instructions for each length.