CSI DSP 接口说明手册

Release v4.2

2022 年 01 月 18 日

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:

第一章　CSI DSP 软件库

1.1　简介

这份手册描述的是 CSI DSP 软件库, 是一套用于玄铁处理器（E804/I805/E906/E907）的通用信号处理函。

库内的各个函数可以分类为:

- 基本数学函数
- 快速数学函数
- 复数函数
- 滤波函数
- 矩阵函数
- 变换函数
- 电机控制函数
- 统计函数
- 辅助函数
- 插值函数

库内的大多数函数都有 Q7，Q15，Q31 和单精度浮点四种版本。

1.2　如何使用库

lib 目录内提供了预编译的版本，如 E906/E907 为:

- libcsi_xt900p32_math.a

函数库的函数声明在头文件 `csi_math.h` 中，`csi_math.h` 源码放在 include 目录中。

在应用程序中包括 `csi_math.h` 头文件，就可以直接调用 DSP 库函数；链接的时候指定应用对应的库版本，就可以将库函数链接进应用程序。

注意 原来以 `csky_` 开头的命名方式将被以 `csi_` 开头的方式取代，并且，不同版本的库都会使用这种命名方式。因此，推荐使用以 `csi` 的开头的命名方式，以 `csky_` 开头的命名方式在下一个版本不再支持。

1.3 不同版本说明

不同指令集版本的库对比如下：

库版本	对应指令集	优化范围
E804	CSKY DSP	定点函数
I805	CSKY VDSP2	定点函数
C860	CSKY VDSP2	所有函数
E906/E907	RISCV DSP	定点函数

说明 个别函数因为特殊需求，只为特定处理器做了定制实现，相关函数各版本库支持情况如下 (其中 Y 表示支持，N 表示不支持):

函数名	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 示例

CSI SDK 里面带了一些示例演示如何使用库函数。

第二章　基本数学函数

2.1　向量绝对值

2.1.1　函数

- *csi_abs_f32* : 浮点向量绝对值.
- *csi_abs_q15* : Q15 向量绝对值.
- *csi_abs_q31* : Q31 向量绝对值.
- *csi_abs_q7* : Q7 向量绝对值.

2.1.2　简要说明

向量的每个元素取绝对值.

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

这些函数可以在原值上做计算，也就是说，允许源和目的向量指针指向相同地址。

为浮点，Q7，Q15，Q31 每种类型都提供了不同的函数。

2.1.3　函数说明

2.1.3.1　csi_abs_f32

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

参数:

`*pSrc`: 指向输入向量

`*pDst`: 指向输出向量

`blockSize`: 向量中的元素数

返回值:

无

2.1.3.2 csi_abs_q15

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

参数:

*pSrc: 指向输入向量

*pDst: 指向输出向量

blockSize: 向量中的元素数

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. Q15 的值为-1 (0x8000) 时会饱和为最大的正数值 0x7FFF.

2.1.3.3 csi_abs_q31

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

参数:

*pSrc: 指向输入向量

*pDst: 指向输出向量

blockSize: 向量中的元素数

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法。Q31 的值为-1 (0x80000000) 时会饱和为最大的正数值 0x7FFFFFFF。

2.1.3.4 csi_abs_q7

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

参数:

*pSrc: 指向输入向量

*pDst: 指向输出向量

blockSize: 向量中的元素数

返回值:

无

最佳性能的条件:

输入和输出 buffer 都是 32 位对齐.

缩放和溢出时的行为:

函数使用饱和算法。Q7 的值为 -1 (0x80) 时会饱和为最大的正数值 0x7F 。

2.2 向量绝对值最大值

2.2.1 函数

- *csi_abs_max_q15* : Q15 向量绝对值最大值.
- *csi_abs_max_q31* : Q31 向量元素绝对值最大值.

2.2.2 简要说明

向量的每个元素取绝对值，然后返回绝对值中的最大值（该函数仅支持 E804/I805 版本）。

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

为 Q15，Q31 两种类型提供了不同的函数。

2.2.3 函数说明

2.2.3.1 csi_abs_max_q15

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

参数:

`*pSrc`: 指向输入向量
`*pDst`: 指向最大值
`blockSize`: 向量中的元素数

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. Q15 的值为-1 (0x8000) 时会饱和为最大的正数值 0x7FFF.

2.2.3.2 csi_abs_max_q31

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

参数:

`*pSrc`: 指向输入向量
`*pDst`: 指向最大值
`blockSize`: 向量中的元素数

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法。Q31 的值为-1 (0x80000000) 时会饱和为最大的正数值 0x7FFFFFFF。

2.3 向量加法

2.3.1 函数

- *csi_add_f32* : 浮点向量加法.
- *csi_add_q15* : Q15 向量加法.
- *csi_add_q31* : Q31 向量加法.
- *csi_add_q7* : Q7 向量加法.

2.3.2 简要说明

两个向量的元素逐个相加.

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

为浮点，Q7，Q15，Q31 每种类型都提供了不同的函数.

2.3.3 函数说明

2.3.3.1 csi_add_f32

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

参数:

`*pSrcA`: 指向第一个输入向量
`*pSrcB`: 指向第二个输入向量
`*pDst`: 指向输出向量
`blockSize`: 向量中的元素数

返回值:

无

2.3.3.2 csi_add_q15

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

参数:

`*pSrcA`: 指向第一个输入向量
`*pSrcB`: 指向第二个输入向量
`*pDst`: 指向输出向量
`blockSize`: 向量中的元素数

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 返回值超出 Q15 的最大范围 [0x8000 0x7FFF] 时会被饱和处理.

2.3.3.3 csi_add_q31

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

参数:

`*pSrcA`: 指向第一个输入向量
`*pSrcB`: 指向第二个输入向量
`*pDst`: 指向输出向量
`blockSize`: 向量中的元素数

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 返回值超出 Q31 的最大范围 [0x80000000 0x7FFFFFFF] 时会被饱和处理.

2.3.3.4 csi_add_q7

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

参数:

`*pSrcA`: 指向第一个输入向量
`*pSrcB`: 指向第二个输入向量
`*pDst`: 指向输出向量
`blockSize`: 向量中的元素数

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 返回值超出 Q7 的最大范围 [0x80 0x7F] 时会被饱和处理.

2.4 向量点积

2.4.1 函数

- *csi_dot_prod_f32* : 浮点向量的点积.
- *csi_dot_prod_q15* : Q15 向量的点积.
- *csi_dot_prod_q31* : Q31 向量的点积.
- *csi_dot_prod_q7* : Q7 向量点积.
- *csi_dot_prod_u64xu8* : Uint64_t, Uint8_t 向量点积.

2.4.2 简要说明

计算两个向量的点积. 向量的每个元素相乘，然后累加（csi_dot_prod_u64xu8 仅支持 I805）。

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

为浮点，Q7，Q15，Q31 每种类型都提供了不同的函数.

2.4.3 函数说明

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)
```

参数:

`*pSrcA`: 指向第一个输入向量
`*pSrcB`: 指向第二个输入向量
`blockSize`: 向量中的元素数量
`*result`: 输出的返回结果

返回值:

无

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)
```

参数:

*pSrcA: 指向第一个输入向量

*pSrcB: 指向第二个输入向量

blockSize: 向量中的元素数量

*result: 输出的返回结果

返回值:

无

缩放和溢出时的行为:

中间步骤的乘法用的是 1.15 x 1.15 = 2.30 格式，然后乘法结果累加成一个 64 位 34.30 格式定点数。累加结果有 33 个保护位，相加的时候不需要使用饱和算法，因为不会出现溢出. 返回结果是 34.30 格式.

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)
```

参数:

*pSrcA: 指向第一个输入向量

*pSrcB: 指向第二个输入向量

blockSize: 向量中的元素数量

*result: 输出的返回结果

返回值:

无

缩放和溢出时的行为:

中间步骤的乘法用的是 1.31 x 1.31 = 2.62 格式，并且通过丢弃低 14 位截断成 2.48. 2.48 的乘法结果累加成一个 64 位 16.48 格式定点数。累加结果有 15 个保护位，相加的时候不需要使用饱和算法，因为不会出现溢出. 返回结果是 16.48 格式.

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)
```

参数:

*pSrcA: 指向第一个输入向量

*pSrcB: 指向第二个输入向量

blockSize: 向量中的元素数量

*result: 输出的返回结果

返回值:

无

缩放和溢出时的行为:

中间步骤的乘法用的是 1.7 x 1.7 = 2.14 格式，然后乘法结果累加成一个 18.14 格式定点数。累加结果有 17 个保护位，相加的时候不需要使用饱和算法，因为不会出现溢出. 返回结果是 18.14 格式.

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)
```

参数:

`*pSrcA`: 指向第一个输入向量

`*pSrcB`: 指向第二个输入向量

`blockSize`: 向量中的元素数量

`*result`: 输出的返回结果

返回值:

无

缩放和溢出时的行为:

中间步骤的乘法用的是 64 位累加器，累加结果没有保护位，且未使用饱和算法，使用时需要限制输入值的范围，以防溢出.

2.5 向量乘法

2.5.1 函数

- *csi_mult_f32* : 浮点向量乘法.
- *csi_mult_q15* : Q15 向量乘法.
- *csi_mult_q31* : Q31 向量乘法.
- *csi_mult_q7* : Q7 向量乘法.
- *csi_mult_rnd_q15* : Q15 带舍入向量乘法.
- *csi_mult_q15xq31_sht* : Q15xQ31 带移位向量相乘.

2.5.2 简要说明

两个向量的元素逐个相乘（csi_mult_rnd_q15 接口不支持 E906/E907，csi_mult_q15xq31_sht 仅支持 I805）。

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

为浮点，Q7，Q15，Q31 每种类型都提供了不同的函数.

2.5.3 函数说明

2.5.3.1 csi_mult_f32

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

参数:

`*pSrcA`: 指向第一个输入向量
`*pSrcB`: 指向第二个输入向量
`*pDst`: 指向输出向量
`blockSize`: 每个向量的元素数量

返回值:

无

2.5.3.2 csi_mult_q15

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

参数:

*pSrcA: 指向第一个输入向量
*pSrcB: 指向第二个输入向量
*pDst: 指向输出向量
blockSize: 每个向量的元素数量

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 结果超出 Q15 的最大范围 [0x8000 0x7FFF] 时会被饱和处理.

2.5.3.3 csi_mult_q31

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

参数:

*pSrcA: 指向第一个输入向量
*pSrcB: 指向第二个输入向量
*pDst: 指向输出向量
blockSize: 每个向量的元素数量

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 结果超出 Q31 的最大范围 [0x80000000 0x7FFFFFFF] 时会被饱和处理.

2.5.3.4 csi_mult_q7

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

参数:

*pSrcA: 指向第一个输入向量
*pSrcB: 指向第二个输入向量
*pDst: 指向输出向量
blockSize: 每个向量的元素数量

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 结果超出 Q7 的最大范围 [0x80 0x7F] 时会被饱和处理.

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)
```

参数:

*pSrcA: 指向第一个输入向量

*pSrcB: 指向第二个输入向量

*pDst: 指向输出向量

blockSize: 每个向量的元素数量

返回值:

无

缩放和溢出时的行为:

函数使用了饱和算法。输出值超出了 Q15 的允许范围 [0x8000 0x7FFF] 就会被饱和，且在饱和之前，结果会被舍入，舍入方式为向正无穷舍入。

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)
```

参数:

*pSrcA: 指向第一个输入向量

*pSrcB: 指向第二个输入向量和输出向量和输出结果

shiftVale: 输出结果移位值，需大于等于 0

blockSize: 每个向量的元素数量

返回值:

无

缩放和溢出时的行为:

函数使用了饱和算法。输出值超出了 Q31 的允许范围 [0x80000000 0x7FFFFFFF] 就会被饱和，且在饱和之前，需要对结果右移 shiftValue。

2.6 向量相反数

2.6.1 函数

- *csi_negate_f32* : 浮点向量的所有元素取相反数.
- *csi_negate_q15* : Q15 向量的所有元素取相反数.
- *csi_negate_q31* : Q31 向量的所有元素取相反数.
- *csi_negate_q7* : Q7 向量的所有元素取相反数.

2.6.2 简要说明

向量的所有元素取相反数。

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

这些函数可以在原值上做计算，也就是说，允许源和目的向量指针指向相同地址。

为浮点，Q7，Q15，Q31 每种类型都提供了不同的函数。

2.6.3 函数说明

2.6.3.1 csi_negate_f32

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

参数:

`*pSrc`: 指向输入向量
`*pDst`: 指向输出向量
`blockSize`: 向量中的元素数量

返回值:

无

2.6.3.2 csi_negate_q15

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

参数:

`*pSrc`: 指向输入向量
`*pDst`: 指向输出向量
`blockSize`: 向量中的元素数量

返回值:

无

最佳性能的条件

输入和输出 buffer 都是 32 位对齐

缩放和溢出时的行为:

函数使用饱和算法. Q15 的值为 -1 (0x8000) 时会饱和为最大的正数值 0x7FFF.

2.6.3.3 csi_negate_q31

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

参数:

`*pSrc`: 指向输入向量
`*pDst`: 指向输出向量
`blockSize`: 向量中的元素数量

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. Q31 的值为 -1 (0x80000000) 时会饱和为最大的正数值 0x7FFFFFFF.

2.6.3.4 csi_negate_q7

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

参数:

`*pSrc`: 指向输入向量
`*pDst`: 指向输出向量
`blockSize`: 向量中的元素数量

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. Q7 的值为 -1 (0x80) 时会饱和为最大的正数值 0x7F.

2.7 向量偏移

2.7.1 函数

- *csi_offset_f32* : 浮点向量添加一个常数偏移量
- *csi_offset_q15* : Q15 向量添加一个常数偏移量
- *csi_offset_q31* : Q31 向量添加一个常数偏移量
- *csi_offset_q7* : Q7 向量添加一个常数偏移量.

2.7.2 简要说明

向量的每个元素添加一个常数偏移量。

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

这些函数可以在原值上做计算，也就是说，允许源和目的向量指针指向相同地址。

为浮点，Q7，Q15，Q31 每种类型都提供了不同的函数.

2.7.3 函数说明

2.7.3.1 csi_offset_f32

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

参数:

`*pSrc`: 指向输入向量
`offset`: 添加的偏移量
`*pDst`: 指向输出向量
`blockSize`: 向量中的元素数

返回值:

无

2.7.3.2 csi_offset_q15

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

参数:

*pSrc: 指向输入向量
offset: 添加的偏移量
*pDst: 指向输出向量
blockSize: 向量中的元素数

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 结果超出 Q15 的最大范围 [0x8000 0x7FFF] 时会被饱和处理.

2.7.3.3 csi_offset_q31

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

参数:

*pSrc: 指向输入向量
offset: 添加的偏移量
*pDst: 指向输出向量
blockSize: 向量中的元素数

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 结果超出 Q31 的最大范围 [0x80000000 0x7FFFFFFF] 时会被饱和处理.

2.7.3.4 csi_offset_q7

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

参数:

*pSrc: 指向输入向量
offset: 添加的偏移量
*pDst: 指向输出向量
blockSize: 向量中的元素数

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 结果超出 Q7 的最大范围 [0x80 0x7F] 时会被饱和处理.

2.8 向量缩放

2.8.1 函数

- *csi_scale_f32* : 浮点向量缩放.
- *csi_scale_q15* : Q15 向量缩放.
- *csi_scale_q31* : Q31 向量缩放.
- *csi_scale_q7* : Q7 向量缩放.

2.8.2 简要说明

将向量乘以标量值. 对浮点数据来说，算法如下:

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

在定点函数 Q7, Q15 和 Q31 中, scale 表现为一个分数乘法 scaleFract 和一个算术移位 shift. 该偏移允许缩放操作的增益超过 1.0:

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

应用于定点数据的总比例因子是

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

这些函数可以在原值上做计算，也就是说，允许源和目的向量指针指向相同地址。

2.8.3 函数说明

2.8.3.1 csi_scale_f32

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

参数:

`*pSrc`: 指向输入向量

`scale`: 缩放比例

`*pDst`: 指向输出向量

`blockSize`: 向量中的元素数

返回值:

无

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)
```

参数:

*pSrc: 指向输入向量
scaleFract: 小数部分的比例值
shift: 将结果移位的位数
*pDst: 指向输出向量
blockSize: 向量中的元素数

返回值:

无

缩放和溢出时的行为:

输入数据 pSrc 和 scaleFract 是 1.15 格式. 相乘的中间结果是 2.30 格式，然后做饱和移位到 1.15 格式.

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)
```

参数:

*pSrc: 指向输入向量
scaleFract: 小数部分的比例值
shift: 将结果移位的位数
*pDst: 指向输出向量
blockSize: 向量中的元素数

返回值:

无

缩放和溢出时的行为:

输入数据 pSrc 和 scaleFract 是 1.31 格式. 相乘的中间结果是 2.62 格式，然后做饱和移位到 1.31 格式.

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)
```

参数:

`*pSrc`: 指向输入向量
`scaleFract`: 小数部分的比例值
`shift`: 将结果移位的位数
`*pDst`: 指向输出向量
`blockSize`: 向量中的元素数

返回值:

无

缩放和溢出时的行为:

输入数据 pSrc 和 scaleFract 是 1.7 格式. 相乘的中间结果是 2.14 格式，然后做饱和移位到 1.7 格式.

2.9 向量移位

2.9.1 函数

- *csi_shift_q15* : Q15 向量的所有元素移位指定位数
- *csi_shift_q31* : Q31 向量的所有元素移位指定位数
- *csi_shift_q7* : Q7 向量的所有元素移位指定位数

2.9.2 简要说明

将定点向量的元素移位指定的位数. 为浮点，Q7，Q15，Q31 每种类型都提供了不同的函数. 使用的算法如下:

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

如果 shift 是正数，则向量的元素向左移位. 如果 shift 是负数，则向量的元素向右移位.

这些函数可以在原值上做计算，也就是说，允许源和目的向量指针指向相同地址.

2.9.3 函数说明

2.9.3.1 csi_shift_q15

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

参数:

`*pSrc`: 指向输入向量
`shiftBits`: 移位的数量. 正数是左移，负数是右移.
`*pDst`: 指向输出向量
`blockSize`: 向量中元素的数量

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 结果超出 Q15 的最大范围 [0x8000 0x7FFF] 时会被饱和处理.

2.9.3.2 csi_shift_q31

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

参数:

*pSrc: 指向输入向量
shiftBits: 移位的数量. 正数是左移，负数是右移
*pDst: 指向输出向量
blockSize: 向量中元素的数量

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 结果超出 Q31 的最大范围 [0x80000000 0x7FFFFFFF] 时会被饱和处理.

2.9.3.3 csi_shift_q7

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

参数:

*pSrc: 指向输入向量
shiftBits: 移位的数量. 正数是左移，负数是右移
*pDst: 指向输出向量
blockSize: 向量中元素的数量

返回值:

无

最佳性能的条件

输入和输出 buffer 都是 32 位对齐

缩放和溢出时的行为:

函数使用饱和算法. 结果超出 Q7 的最大范围 [0x80 0x7F] 时会被饱和处理.

2.10 向量减法

2.10.1 函数

- *csi_sub_f32* : 浮点向量减法.
- *csi_sub_q15* : Q15 向量减法.
- *csi_sub_q31* : Q31 向量减法.
- *csi_sub_q7* : Q7 向量减法.

2.10.2 简要说明

两个向量的元素逐个相减.

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

为浮点，Q7，Q15，Q31 每种类型都提供了不同的函数.

2.10.3 函数说明

2.10.3.1 csi_sub_f32

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

参数:

`*pSrcA`: 指向第一个输入向量
`*pSrcB`: 指向第二个输入向量
`*pDst`: 指向输出向量
`blockSize`: 向量中元素的数量

返回值:

无

2.10.3.2 csi_sub_q15

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

参数:

`*pSrcA`: 指向第一个输入向量
`*pSrcB`: 指向第二个输入向量
`*pDst`: 指向输出向量
`blockSize`: 向量中元素的数量

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 结果超出 Q15 的最大范围 [0x8000 0x7FFF] 时会被饱和处理.

2.10.3.3 csi_sub_q31

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

参数:

`*pSrcA`: 指向第一个输入向量
`*pSrcB`: 指向第二个输入向量
`*pDst`: 指向输出向量
`blockSize`: 向量中元素的数量

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 结果超出 Q31 的最大范围 [0x80000000 0x7FFFFFFF] 时会被饱和处理.

2.10.3.4 csi_sub_q7

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

参数:

`*pSrcA`: 指向第一个输入向量
`*pSrcB`: 指向第二个输入向量
`*pDst`: 指向输出向量
`blockSize`: 向量中元素的数量

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 结果超出 Q7 的最大范围 [0x80 0x7F] 时会被饱和处理.

2.11　向量求和

2.11.1　函数

- *csi_sum_q15* : Q15 向量求和.

2.11.2　简要说明

向量的每个元素累加求和（该函数仅支持 I805）。

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

为 Q15 类型都提供了函数.

2.11.3　函数说明

2.11.3.1　csi_sum_q15

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

参数:

`*pSrcA`: 指向输入向量

`*pDst`: 指向输出地址

`blockSize`: 向量中元素的数量

返回值:

无

缩放和溢出时的行为:

函数使用了 64 位累加器，有 48 个保护位，因此不会溢出. 不需要进行饱和处理.

第三章　复数函数

这组函数操作复数向量。复数向量的元素以交错的方式保存 (real, imag, real, imag, …). 接口函数中，指定的样本数量是复数元素的个数；向量实际包括了两倍数量的数值。

3.1　复数共轭

3.1.1　函数

- *csi_cmplx_conj_f32* : 浮点复数共轭
- *csi_cmplx_conj_q15* : Q15 复数共轭
- *csi_cmplx_conj_q31* : Q31 复数共轭

3.1.2　简要说明

复数向量的所有元素共轭

pSrc 指向源数据，pDst 指向结果写入的目的地址. numSamples 指定复数元素的个数，复数数据是交错方式保存的 (real, imag, real, imag, …). 每个向量一共有 2*numSamples 个值.

使用的算法如下:

```
for (n=0; n<numSamples; n++) {
    pDst[(2 * n) + 0)] = pSrc[(2 * n) + 0];      // real part
    pDst[(2 * n) + 1)] = -pSrc[(2 * n) + 1];     // imag part
}
```

为浮点，Q15 和 Q31 三种类型都提供了不同的函数.

3.1.3　函数说明

3.1.3.1　csi_cmplx_conj_f32

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

参数:

*pSrc: 指向输入向量

*pDst: 指向输出向量

numSamples: 向量中的复数元素个数

返回值:

无

3.1.3.2 csi_cmplx_conj_q15

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

参数:

*pSrc: 指向输入向量

*pDst: 指向输出向量

numSamples: 向量中的复数元素个数

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. Q15 的值为 -1 (0x8000) 时会饱和为最大的正数值 0x7FFF.

3.1.3.3 csi_cmplx_conj_q31

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

参数:

*pSrc: 指向输入向量

*pDst: 指向输出向量

numSamples: 向量中的复数元素个数

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. Q31 的值为 -1 (0x80000000) 时会饱和为最大的正数值 0x7FFFFFFF.

3.2 复数点积

3.2.1 函数

- *csi_cmplx_dot_prod_f32* : 浮点复数点积
- *csi_cmplx_dot_prod_q15* : Q15 复数点积
- *csi_cmplx_dot_prod_q31* : Q31 复数点积

3.2.2 简要说明

计算两个复数向量的点积. 向量的元素逐个相乘，然后累加.

pSrcA 指向第一个复数输入向量，pSrcB 指向第二个复数输入向量. numSamples 指定复数元素的个数，复数数据是交错方式保存的 (real, imag, real, imag, ···). 每个向量一共有 2*numSamples 个值.

使用的算法如下:

```
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];
}
```

为浮点，Q15 和 Q31 三种类型都提供了不同的函数.

3.2.3 函数说明

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)
```

参数:

`*pSrcA`: 指向第一个输入向量

`*pSrcB`: 指向第二个输入向量

`numSamples`: 向量中的复数元素数量

`*realResult`: 结果的实部

`*imagResult`: 结果的虚部

返回值:

无

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)
```

参数:

*pSrcA: 指向第一个输入向量

*pSrcB: 指向第二个输入向量

numSamples: 向量中的复数元素数量

*realResult: 结果的实部

*imagResult: 结果的虚部

返回值:

无

缩放和溢出时的行为:

函数的实现使用了一个内部的 64 位累加器. 1.15 格式和 1.15 格式相乘的中间结果用的是全精度, 产生 2.30 格式的结果. 64 位累加器将结果累加位 34.30 格式的值. 最后, 累加结果转换为 8.24 格式. 返回结果的 realResult 和 imagResult 是 8.24 格式.

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)
```

参数:

*pSrcA: 指向第一个输入向量

*pSrcB: 指向第二个输入向量

numSamples: 向量中的复数元素数量

*realResult: 结果的实部

*imagResult: 结果的虚部

返回值:

无

缩放和溢出时的行为:

函数的实现使用了一个内部的 64 位累加器. 1.31 格式和 1.31 格式相乘的结果是 64 位精度，移位成 16.48 格式. 中间结果的实部和虚部累加用的都是 16.48 格式. 只要 numSamples 少于 32768，加法就不会溢出，也就不需要饱和操作. 返回结果 realResult 和 imagResult 是 16.48 格式. 不需要输入向下缩放.

3.3 复数幅度

3.3.1 函数

- *csi_cmplx_mag_f32* : 浮点向量幅度
- *csi_cmplx_mag_q15* : Q15 复数幅度
- *csi_cmplx_mag_q31* : Q31 复数幅度

3.3.2 简要说明

计算复数向量的每个元素的幅度.

pSrc 指向源数据，pDst 指向结果写入的地址. numSamples 指定输入向量的复数元素个数，复数数据是交错方式保存的 (real, imag, real, imag, ···). 输入向量一共有 2*numSamples 个值; 输出向量一共有 numSamples 个值.

使用的算法如下:

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

为浮点，Q15 和 Q31 三种类型都提供了不同的函数.

3.3.3 函数说明

3.3.3.1 csi_cmplx_mag_f32

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

参数:

`*pSrc`: 指向复数输入向量

`*pDst`: 指向输出向量

`numSamples`: 输入向量的复数元素个数

返回值:

无

3.3.3.2 csi_cmplx_mag_q15

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

参数:

*pSrc: 指向复数输入向量

*pDst: 指向输出向量

numSamples: 输入向量的复数元素个数

返回值:

无

缩放和溢出时的行为:

函数实现 1.15 和 1.15 的乘法，最后输出转换成 2.14 格式。当实部和虚部都是 0x8000 时，中间过程可能溢出。

3.3.3.3 csi_cmplx_mag_q31

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

参数:

*pSrc: 指向复数输入向量

*pDst: 指向输出向量

numSamples: 输入向量的复数元素个数

返回值:

无

缩放和溢出时的行为:

函数实现 1.31 和 1.31 乘法，最后输出的结果转换为 2.30 格式. 输入不需要向下缩放.

3.4　复数幅度平方

3.4.1　函数

- *csi_cmplx_mag_squared_f32* : 浮点复数幅度平方
- *csi_cmplx_mag_squared_q15* : Q15 复数幅度平方
- *csi_cmplx_mag_squared_q31* : Q31 复数幅度平方

3.4.2　简要说明

计算复数向量元素的幅度平方.

pSrc 指向源数据，pDst 指向结果写入的地址. numSamples 指定复数元素的个数，复数数据是交错方式保存的 (real, imag, real, imag, ···). 输入向量总共有 2*numSamples 个值; 输出向量总共有 numSamples 个值.

使用的算法如下:

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

为浮点，Q15 和 Q31 三种类型都提供了不同的函数.

3.4.3　函数说明

3.4.3.1　csi_cmplx_mag_squared_f32

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

参数:

`*pSrc`: 指向输入的复数向量

`*pDst`: 指向输出的向量

`numSamples`: 输入向量中的复数数量

返回值:

无

3.4.3.2　csi_cmplx_mag_squared_q15

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

参数:

*pSrc: 指向输入的复数向量

*pDst: 指向输出的向量

numSamples: 输入向量中的复数数量

返回值:

无

缩放和溢出时的行为:

函数实现了 1.15 和 1.15 的乘法，最后将结果转换为 3.13 格式输出.

3.4.3.3 csi_cmplx_mag_squared_q31

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

参数:

*pSrc: 指向输入的复数向量

*pDst: 指向输出的向量

numSamples: 输入向量中的复数数量

返回值:

无

缩放和溢出时的行为:

函数实现了 1.31 和 1.31 的乘法，最后转换成 3.29 格式输出. 输入不需要向下缩放.

3.5 复数与复数相乘

3.5.1 函数

- *csi_cmplx_mult_cmplx_f32* : 浮点复数乘法
- *csi_cmplx_mult_cmplx_q15* : Q15 复数相乘
- *csi_cmplx_mult_cmplx_q31* : Q31 复数相乘

3.5.2 简要说明

将一个复向量与另一个复向量相乘，并生成复数结果. 复数向量中数据是交错方式保存的 (real, imag, real, imag, ···). 参数 numSamples 表示复数元素的数量. 复数向量总共有 2*numSamples 个值

使用的算法如下:

```
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];
}
```

为浮点，Q15 和 Q31 三种类型都提供了不同的函数.

3.5.3 函数说明

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)
```

参数:

`*pSrcA`: 指向第一个输入向量

`*pSrcB`: 指向第二个输入向量

`*pDst`: 指向输出向量

`numSamples`: 向量中的复数元素个数

返回值:

无

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)
```

参数:

*pSrcA: 指向第一个输入向量

*pSrcB: 指向第二个输入向量

*pDst: 指向输出向量

numSamples: 向量中的复数元素个数

返回值:

无

缩放和溢出时的行为:

函数实现了 1.15 和 1.15 乘法，最后的结果转换为 3.13 格式输出.

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)
```

参数:

*pSrcA: 指向第一个输入向量

*pSrcB: 指向第二个输入向量

*pDst: 指向输出向量

numSamples: 向量中的复数元素个数

返回值:

无

缩放和溢出时的行为:

函数实现了 1.31 和 1.31 乘法，最后结果转换为 3.29 格式输出. 输入不需要向下缩放.

3.6 复数与实数相乘

3.6.1 函数

- *csi_cmplx_mult_real_f32* : 浮点的复数和实数相乘
- *csi_cmplx_mult_real_q15* : Q15 复数和实数相乘
- *csi_cmplx_mult_real_q31* : Q31 复数和实数相乘

3.6.2 简要说明

将一个复数向量与一个实数向量相乘，并生成一个复数向量. 复数向量数据是交错方式保存的 (real, imag, real, imag, ···). 参数 numSamples 表示处理的复数元素数量。复数向量总共有 2*numSamples 个值，实数向量总共有 numSamples 个值。

使用的算法如下:

```
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];
}
```

为浮点，Q15 和 Q31 三种类型都提供了不同的函数.

3.6.3 函数说明

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)
```

参数:

`*pSrcCmplx`: 指向输入的复数向量

`*pSrcReal`: 指向输入的实数向量

`*pCmplxDst`: 指向输出的复数向量

`numSamples`: 向量中的元素数量

返回值:

无

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)
```

参数:

*pSrcCmplx: 指向输入的复数向量

*pSrcReal: 指向输入的实数向量

*pCmplxDst: 指向输出的复数向量

numSamples: 向量中的元素数量

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 结果超出 Q15 的最大范围 [0x8000 0x7FFF] 时会被饱和处理.

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)
```

参数:

*pSrcCmplx: 指向输入的复数向量

*pSrcReal: 指向输入的实数向量

*pCmplxDst: 指向输出的复数向量

numSamples: 向量中的元素数量

返回值:

无

缩放和溢出时的行为:

函数使用饱和算法. 结果超出 Q15 的最大范围 [0x80000000 0x7FFFFFFF] 时会被饱和处理.

第四章　电机控制函数

4.1　PID 电机控制

4.1.1　函数

- *csi_pid_f32* : 浮点 PID 控制的处理函数
- *csi_pid_q31* : Q31 PID 控制的处理函数
- *csi_pid_q15* : Q15 PID 控制的处理函数
- *csi_pid_init_f32* : 浮点 PID 控制的初始化函数
- *csi_pid_init_q31* : Q31 PID 控制的初始化函数
- *csi_pid_init_q15* : Q15 PID 控制的初始化函数
- *csi_pid_reset_f32* : 浮点 PID 控制的重置函数
- *csi_pid_reset_q31* : Q31 PID 控制的重置函数
- *csi_pid_reset_q15* : Q15 PID 控制的重置函数

4.1.2　简要说明

比例积分微分 (PID) 控制器是一种通用的反馈环控制机制，广泛应用在工业控制系统中。PID 控制器是最常见的反馈控制器类型.

为 Q15，Q31，浮点类型实现了不同的 PID 控制器的函数。函数每次只操作一个样本数据，并返回一个处理结果 S 指向一个 PID 控制数据结构的实例. in 是输入的样本数据，函数返回输出结果。

算法:

```
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
```

其中 Kp 是比例常数, Ki 是积分常数，Kd 是微分常数

PID 控制器计算测量输出和参考输入之间的差值作为“误差”值。控制器会通过调节控制输入，来最小化“误差”值。比例值决定对当前误差反应的程度，积分值根据最近累积的“误差”和做反应，微分值对“误差”变化的速率做反应。

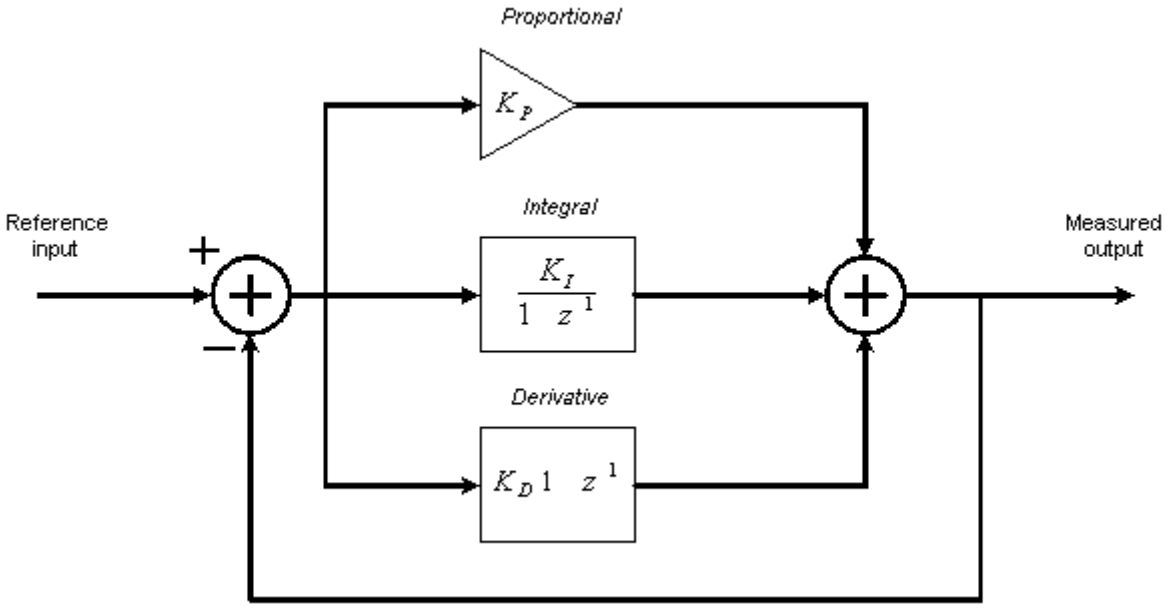


图 4.1: 比例积分微分控制器

结构体实例

PID 控制器的增益 A0, A1, A2 和状态变量都保存在数据结构实例中。每个 PID 控制都必须定义一个单独结构体实例。为支持的 Q15，Q31 和浮点这三种不同数据类型，分别提供了结构体实例的定义。

重置函数

也为不同的数据类型提供了不同的重置函数，用来清除状态。

初始化函数

为不同的数据类型提供了相关的初始化函数。初始化函数处理以下操作:

- 根据 Kp，Ki，Kd 增益，初始化 A0, A1, A2 增益.
- 清零状态 buffer 中的值.

结构体实例不能放进只读数据段，推荐用初始化函数处理。

定点行为

使用定点数版本的 PID 控制器函数时必须注意、特别是，在每个函数中使用的累加器的溢出和饱和行为，必须考虑到。参考特定函数的特定使用文档。

4.1.3　函数说明

4.1.3.1　csi_pid_f32

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

参数:

*S: PID 控制结构的实例

in: 输入处理的样本

返回值:

输出的样本处理结果

4.1.3.2 csi_pid_q31

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

参数:

*S: PID 控制结构的实例

in: 输入处理的样本

返回值:

输出的样本处理结果

缩放和溢出时的行为:

函数实现中使用了一个内部的 64 位累加器。累加器使用 2.62 格式，维护了中间乘法结果的完整精度，但是只提供了一个守护位。因此，累加器的结果如果溢出会折回（覆盖掉符号位）。由于总共有四个数相加，为了防止完全异常（连续溢出两个位），输入信号必须向下缩放两个位。在所有的乘法结果相加后，2.62 格式的累加器截断为 1.32 的格式，然后再饱和成 1.31 的格式。

4.1.3.3 csi_pid_q15

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

参数:

*S: PID 控制结构的实例

in: 输入处理的样本

返回值:

输出的样本处理结果

缩放和溢出时的行为:

函数实现中使用了一个内部的 64 位累加器。增益和状态变量都是 1.15 格式表示，相乘的结果是 2.30 格式。2.30 格式的中间结果在 64 位累加器上累加为 34.30 格式。这种方式没有内部溢出的风险，并且能保存所有的中间结果的精度。在累加完后，累加器将低 15 位截断为 34.15 格式。最后，累加器做饱和处理，得到 1.15 格式的结果。

4.1.3.4 csi_pid_init_f32

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

参数:

`*S`: PID 控制结构的实例

`resetStateFlag`: 重置状态的标志位. 0 = 不改变状态；1 = 重置状态

返回值:

无

简要说明:

resetStateFlag 指定是否重置状态为 0。

函数使用比例增益 (Kp), 积分增益 (Ki) 和微分增益 (Kd) 计算结构体字段: A0, A1 A2 ， 并且将状态变量全部置为 0。

4.1.3.5 csi_pid_init_q31

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

参数:

`*S`: PID 控制结构的实例

`resetStateFlag`: 重置状态的标志位. 0 = 不改变状态；1 = 重置状态

返回值:

无

简要说明:

resetStateFlag 指定是否重置状态为 0。

函数使用比例增益 (Kp), 积分增益 (Ki) 和微分增益 (Kd) 计算结构体字段: A0, A1 A2 ， 并且将状态变量全部置为 0。

4.1.3.6 csi_pid_init_q15

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

参数:

*S: PID 控制结构的实例

resetStateFlag: 重置状态的标志位. 0 = 不改变状态；1 = 重置状态

返回值:

无

简要说明:

resetStateFlag 指定是否重置状态为 0。

函数使用比例增益 (Kp), 积分增益 (Ki) 和微分增益 (Kd) 计算结构体字段: A0, A1 A2 ， 并且将状态变量全部置为 0。

4.1.3.7 csi_pid_reset_f32

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

参数:

*S: PID 控制结构的实例

返回值:

无

简要说明:

函数重置状态 buffer 为 0

4.1.3.8 csi_pid_reset_q31

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

参数:

*S: PID 控制结构的实例

返回值:

无

简要说明:

函数重置状态 buffer 为 0

4.1.3.9 csi_pid_reset_q15

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

参数:

*S: PID 控制结构的实例

返回值:

无

简要说明:

函数重置状态 buffer 为 0

4.2　Clarke 变换

4.2.1　函数

- *csi_clarke_f32* : 浮点 clarke 变换
- *csi_clarke_q31* : Q31 clarke 变换

4.2.2　简要说明

正向 clarke 变换将瞬时定子相转换为时间不变的双坐标向量，即把 abc 坐标转换为 αβ 坐标。一般，clarke 变换使用当前的三相 Ia, Ib 和 Ic 来计算当前的两相正交定子轴 Ialpha 和 Ibeta. 当 Ialpha 如下图叠加在 Ia 上时：

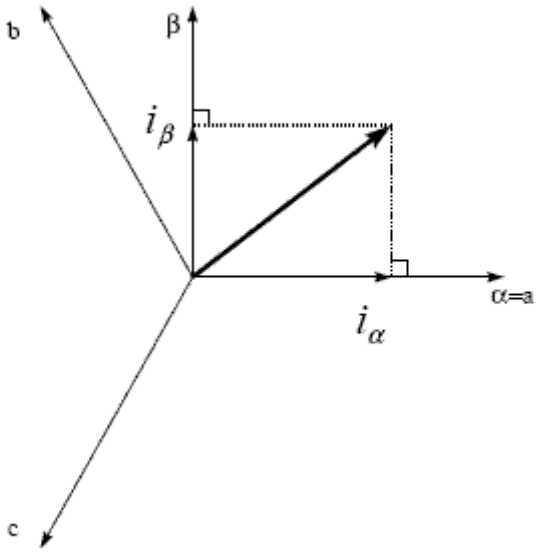


图 4.2: 定子电流空间矢量及其分量 (a,b)

因为 Ia + Ib + Ic = 0, 所以 Ialpha 和 Ibeta 可以只用 Ia 和 Ib 计算出.

函数一次处理一个单独样本，每次调用直接返回处理结果。为 Q31 和浮点数据类型分别提供了不同的函数。

算法

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

其中 Ia 和 Ib 是瞬时定子相，pIalpha 和 pIbeta 是时间不变的两个两相坐标向量

定点行为

使用 Q31 版本 clarke 变换函数需要注意。特别是要考虑，在每个函数内使用的累加器的溢出和饱和行为。具体参考每个函数各自的文档和使用说明。

4.2.3 函数说明

4.2.3.1 csi_clarke_f32

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

参数:

Ia: 输入的三相坐标 a

Ib: 输入的三相坐标 b

*pIalpha: 指向输出的两相正交矢量轴 alpha

*pIbeta: 指向输出的两相正交矢量轴 beta

返回值:

无

4.2.3.2 csi_clarke_q31

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

参数:

Ia: 输入的三相坐标 a

Ib: 输入的三相坐标 b

*pIalpha: 指向输出的两相正交矢量轴 alpha

*pIbeta: 指向输出的两相正交矢量轴 beta

返回值:

无

缩放和溢出时的行为:

函数实现使用了一个内部 32 位累加器. 累加器维持 1.31 格式，丢弃中间乘法结果 2.62 格式的低 32 位。相加用的是饱和计算，所以不会溢出。

4.3 逆向 clarke 变换

4.3.1 函数

- *csi_inv_clarke_f32* : 浮点逆向 clarke 变换
- *csi_inv_clarke_q31* : Q31 逆向 clarke 变换

4.3.2 简要说明

反向 clarke 变换转换时间不变双坐标向量为瞬时定子相，即把 αβ 坐标转换成 abc 坐标。

函数每次操作一个单独的数据样本，并且每次调用直接返回处理结果。库为 Q31 和浮点数据类型分别提供了不同的函数。

算法

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

其中 pIa 和 pIb 是瞬时定子相，Ialpha 和 Ibeta 是时间不变的双坐标向量。

定点行为

使用 Q31 版本 clarke 变换函数需要注意。特别是要考虑，在每个函数内使用的累加器的溢出和饱和行为。具体参考每个函数各自的文档和使用说明。

4.3.3 函数说明

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)
```

参数:

`Ialpha`: 输入的两相正交矢量轴 alpha

`Ibeta`: 输入的两相正交矢量轴 beta

`*pIa`: 输出的三相坐标 a

`*pIb`: 输出的三相坐标 b

返回值:

无

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)
```

参数:

`Ialpha`: 输入的两相正交矢量轴 alpha

`Ibeta`: 输入的两相正交矢量轴 beta

`*pIa`: 输出的三相坐标 a

`*pIb`: 输出的三相坐标 b

返回值:

无

缩放和溢出时的行为:

函数实现使用了一个内部 32 位累加器。累加器维持 1.31 格式，丢弃中间乘法结果 2.62 格式的低 32 位相加用的是饱和计算，所以不会溢出。

4.4　Park 变换

4.4.1　函数

- *csi_park_f32* : 浮点 park 变换
- *csi_park_q31* : Q31 版本的 Park 变换

4.4.2　简要说明

正向 Park 变换将输入的双坐标矢量转换为通量和转矩分量，即把 αβ 坐标转换为 dq 坐标。Park 变换可以将当前的静止 Ialpha 和 Ibeta 参考系，转换到移动参考系，变成定子矢量电流和转子磁链矢量的空间关系。如果我们考虑对齐转子磁链的 d 轴，下面的框图演示了当前的向量和两个参考系之间的关系：

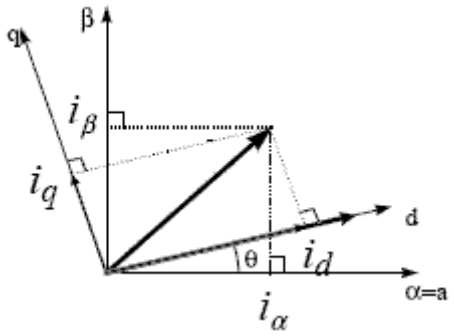


图 4.3: 定子电流空间矢量和它在 (a,b) 的组成，以及在 d,q 旋转参考系

函数每次操作单个的样本数据，并且每次调用都返回处理结果。库为 Q31 和浮点数据类型分别提供不同的函数。

算法

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

其中 Ialpha 和 Ibeta 是定义定子矢量分量，pId 和 pIq 是转子矢量分量，并且 cosVal 和 sinVal 是 θ（转子磁链位置）的余弦和正弦值。

定点行为

使用定点 Park 变换函数需要注意。特别是要考虑，在每个函数内使用的累加器的溢出和饱和行为。具体参考每个函数各自的文档和使用说明。

4.4.3 函数说明

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)
```

参数:

`Ialpha`: 输入的两相向量坐标 alpha

`Ibeta`: 输入的两相向量坐标 beta

`*pId`: 指向输出的转子参考系 d

`*pIq`: 指向输出的转子参考系 q

`sinVal`: 旋转角 θ 的正弦

`cosVal`: 旋转角 θ 的余弦

返回值:

无

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)
```

参数:

`Ialpha`: 输入的两相向量坐标 alpha

`Ibeta`: 输入的两相向量坐标 beta

`*pId`: 指向输出的转子参考系 d

`*pIq`: 指向输出的转子参考系 q

`sinVal`: 旋转角 θ 的正弦

`cosVal`: 旋转角 θ 的余弦

返回值:

无

缩放和溢出时的行为:

函数实现使用了一个内部的 32 位累加器。累加器的是 1.31 格式，丢弃了中间乘法结果的低 31 位。加法和减法的时候用了饱和计算，所以没有溢出的风险。

4.5　逆向 park 变换

4.5.1　函数

- *csi_inv_park_f32* : 浮点逆向 Park 变换
- *csi_inv_park_q31* : Q31 版本的逆向 Park 变换

4.5.2　简要说明

逆向 Park 变换将输入的磁链和转矩分量转换为双坐标向量。即把 dq 坐标转换为 αβ 坐标。
函数每次操作单个的样本数据，并且每次调用都返回处理结果。库为 Q31 和浮点数据类型分别提供不同的函数。

算法

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

其中 pIalpha 和 pIbeta 是定子矢量分量, Id 和 Iq 是转子矢量分量，cosVal 和 sinVal 是 θ（转子磁链位置）的余弦和正弦值。

定点行为

使用定点 Park 变换函数需要注意。特别是要考虑，在每个函数内使用的累加器的溢出和饱和行为。具体参考每个函数各自的文档和使用说明。

4.5.3　函数说明

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)
```

参数:

`Id`: 转子参考系的输入坐标 d
`Iq`: 转子参考系的输入坐标 q
`*pIalpha`: 两相正交向量轴输出 alpha
`*pIbeta`: 两相正交向量轴输出 beta
`sinVal`: 旋转角 θ 的正弦
`cosVal`: 旋转角 θ 的余弦

返回值:

无

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)
```

参数:

Id: 转子参考系的输入坐标 d

Iq: 转子参考系的输入坐标 q

*pIalpha: 两相正交向量轴输出 alpha

*pIbeta: 两相正交向量轴输出 beta

sinVal: 旋转角 θ 的正弦

cosVal: 旋转角 θ 的余弦

返回值:

无

缩放和溢出时的行为:

函数实现使用了一个内部 32 位累加器。累加器维持 1.31 格式，丢弃中间乘法结果 2.62 格式的低 32 位相加用的是饱和计算，所以不会溢出。

4.6 正弦余弦

4.6.1 函数

- *csi_sin_cos_f32* : 浮点正弦余弦函数
- *csi_sin_cos_q31* : Q31 正弦余弦函数

4.6.2 简要说明

通过一组表的查找和插值，来计算三角正弦和余弦。为 Q31 和浮点数据类型提供了不同的函数。浮点版本的输入是以度为单位，定点 Q31 的输入是缩放映射到 [-180 +180] 度的 [-1 0.9999]。

浮点函数也支持超出一般范围的值。发生这种情况的时候，函数会需要额外的时间来调整输入值到 [-180 180] 的范围.

算法实现基于 360 个表项的线性插值。使用的步骤如下:

1. 计算最接近的整数表索引.
2. 计算输入的小数部分 (fract).
3. 根据索引 index 从正弦表中发现对应的 y0 和索引 index+1 对应的 y1.
4. 正弦值计算 psinVal = y0 + (fract * (y1 - y0)).
5. 根据索引 index 从余弦表中发现对应的 y0 和索引 index+1 对应的 y1.
6. 余弦值计算 pcosVal = y0 + (fract * (y1 - y0)).

4.6.3 函数说明

4.6.3.1 csi_sin_cos_f32

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

参数:

`theta`: 度数输入值

`*pSinVal`: 指向正弦输出结果

`*pCosVal`: 指向余弦输出结果

返回值:

无

4.6.3.2 csi_sin_cos_q31

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

参数:

theta: 度数输入值

*pSinVal: 指向正弦输出结果

*pCosVal: 指向余弦输出结果

返回值:

无

note:

Q31 输入值的范围是 [-1 0.999999]，然后映射到的度数范围是 [-180 179].

第五章　快速数学函数

这组函数提供了一种快速求解正弦，余弦和平方根近似值的方法。与 CSI DSP 库中大多数其他函数不同，快速函数只操作一个值，而不是以数组（向量）为单位。为 Q15，Q31 和浮点数据类型分别提供了不同的函数。

5.1　余弦

5.1.1　函数

- *csi_cos_f32* : 浮点三角余弦的近似值快速算法
- *csi_cos_q15* : Q15 三角余弦的近似值快速算法
- *csi_cos_q31* : Q31 三角余弦的近似值快速算法

5.1.2　简要说明

用查表和插值的方法计算三角余弦函数。为 Q15，Q31 和浮点类型都提供了不同的函数。浮点的版本的输入值用的是弧度，Q15 和 Q31 用的输入是缩放后的 [0 +0.9999]，映射到 [0 2*pi)。定点范围的选值不包括 2*pi，2*pi 会绕回到 0。

算法的实现基于 256 个表项查表和线性插值。步骤如下:

1. 计算表内最接近的整数索引值
2. 计算表索引的小数 (fract) 部分
3. 最后的结果等于 (1.0f - fract) * a + fract * b;

其中

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

5.1.3　函数说明

5.1.3.1　csi_cos_f32

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

参数:

x: 输入的弧度值

返回值:

cos(x).

5.1.3.2 csi_cos_q15

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

参数:

x: 输入的弧度值

返回值:

cos(x).

缩放和溢出时的行为:

Q15 输入值在 [0, 1) 区间，映射到弧度的区间是 [0, 2*pi).

5.1.3.3 csi_cos_q31

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

参数:

x: 输入的弧度值

返回值:

cos(x).

缩放和溢出时的行为:

Q31 输入值的范围是 [0, 1)，映射的范围是 [0, 2*pi).

5.2 正弦

5.2.1 函数

- *csi_sin_f32* : 浮点数三角正弦的近似值快速算法
- *csi_sin_q15* : Q15 浮点数三角正弦的近似值快速算法
- *csi_sin_q31* : Q31 浮点数三角正弦的近似值快速算法

5.2.2 简要说明

用查表和插值的方法计算三角余弦函数。为 Q15，Q31 和浮点类型都提供了不同的函数。浮点的版本的输入值用的是弧度，Q15 和 Q31 用的输入是缩放后的 [0 +0.9999]，映射到 [0 2*pi)。定点范围的选值不包括 2*pi，2*pi 会绕回到 0。

算法的实现基于 256 个表项查表和线性插值。步骤如下:

1. 计算表内最接近的整数索引值
2. 计算表索引的小数 (fract) 部分
3. 最后的结果等于 (1.0f-fract) * a + fract * b;

其中

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

5.2.3 函数说明

5.2.3.1 csi_sin_f32

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

参数:

x: 输入的弧度值

返回值:

sin(x).

5.2.3.2 csi_sin_q15

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

参数:

x: 输入的弧度值

返回值:

sin(x).

缩放和溢出时的行为:

Q15 输入值的范围是 [0，1)，映射的范围是 [0, 2*pi).

5.2.3.3 csi_sin_q31

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

参数:

x: input value in radians.

返回值:

sin(x).

缩放和溢出时的行为:

Q31 输入值的范围是 [0, 1)，映射的范围是 [0, 2*pi).

5.3 平方根

5.3.1 函数

- *csi_sqrt_q15* : Q15 平方根函数.
- *csi_sqrt_q31* : Q31 平方根函数.

5.3.2 简要说明

计算一个数的平方根。为 Q15，Q31 实现了不同的函数。当 CPU 有 FPU 时，指令 fsqrts 被用来计算结果，而当没有 FPU 时，牛顿迭代法被用来计算结果。迭代的算法公式如下:

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

其中 x1 是当前估计值, x0 是上一次估计值, f’(x0) 是 f() 在 x0 点的导数。对于平方根函数，算法简化为:

```
x0 = in/2                          [最初的猜测值]
x1 = 1/2 * ( x0 + in / x0)         [每次迭代的公式]
```

5.3.3 函数说明

5.3.3.1 csi_sqrt_q15

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

参数:

`in`: 输入值 `pOut`: 输入值的平方根

返回值:

如果输入值是正数，函数返回 CSKY_MATH_SUCCESS，如果输入是负数，则返回 CSKY_MATH_ARGUMENT_ERROR. 对负数输入，返回的参数 *pOut = 0.

5.3.3.2 csi_sqrt_q31

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

参数:

`in`: 输入值 `pOut`: 输入值的平方根

返回值:

如果输入值是正数，函数返回 CSKY_MATH_SUCCESS，如果输入是负数，则返回 CSKY_MATH_ARGUMENT_ERROR. 对负数输入，返回的参数 *pOut = 0.

Note

当硬浮点指令 fsqrts 被使用时，函数的精度会从 3 LSB 降到 7 LSB。调用这个函数的相关函数也是如此。

第六章　滤波函数

6.1　直接 I 型 IIR 滤波器

6.1.1　函数

- *csi_biquad_cascade_df1_f32* : 浮点二阶级联滤波器的处理函数
- *csi_biquad_cascade_df1_q15* : Q15 二阶级联滤波器的处理函数
- *csi_biquad_cascade_df1_q31* : Q31 二阶级联滤波器的处理函数
- *csi_biquad_cascade_df1_fast_q15* : Q15 二阶级联滤波器的处理函数
- *csi_biquad_cascade_df1_fast_q31* : Q31 二阶级联滤波器的处理函数
- *csi_biquad_cascade_df1_init_f32* : 浮点二阶级联滤波器的初始化函数
- *csi_biquad_cascade_df1_init_q15* : Q15 二阶级联滤波器的初始化函数
- *csi_biquad_cascade_df1_init_q31* : Q31 二阶级联滤波器的初始化函数

6.1.2　简要说明

这些函数实现了任意阶递归（IIR）滤波器。滤波器是由二阶单元 Biquad 级联实现。为 Q15，Q31 和浮点数据类型都提供了函数。

函数以块单位操作输入输出数据。每次调用函数通过滤波器处理 blockSize 个样本。pSrc 指向输入数组数据，pDst 指向输出数组数据。两个数组都包含 blockSize 个数值.

算法

每个二阶阶段都使用差分方程实现一个二阶滤波:

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

直接 I 型每个阶段使用 5 个系数和 4 个状态变量。

系数 b0, b1 和 b2 与输入信号 x[n] 相乘，并且被称为前馈系数。系数 a1 和 a2 与输出信号 y[n] 相乘，并且被称为反馈系数。

需要注意反馈系数的符号. 有些设计工具使用差分方程:

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

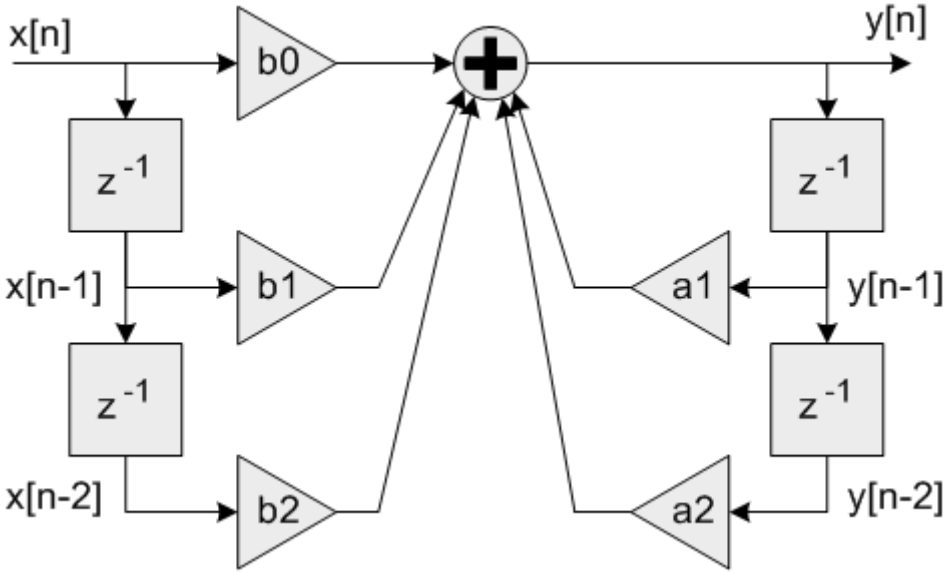


图 6.1: Single Biquad filter stage

这种情况下，反馈系数 a1 和 a2 在使用 CSI DSP 库的时候，必须取相反数。

高阶滤波器通过级联的二阶单元实现。numStages 指定使用了多少二阶阶段。比如，第 8 阶滤波器是 numStages=4 的二阶阶段。

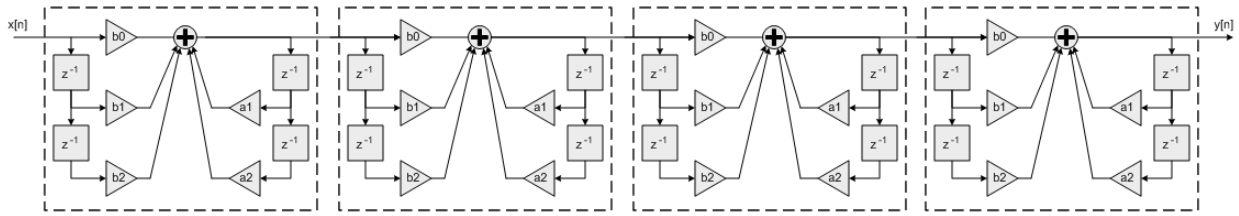


图 6.2: 使用二阶级联的 8 阶滤波器

第 9 阶滤波器可以看做 numStages=5 的二阶阶段，系数配置为第一阶 (b2=0 和 a2=0).

pState 指向状态变量数组. 每个二阶阶段有 4 个状态变量 x[n-1], x[n-2], y[n-1], 和 y[n-2]. 状态变量被排列在 pState :

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

阶段 1 使用最开始的 4 个状态变量，阶段 2 使用接下来的 4 个状态变量，依次类推。状态变量数组总共有 4*numStages 个值。每个块数据处理会更新状态变量，不会更新系数。

结构体实例

滤波器的系数和状态变量都保存在数据结构中。每个滤波器都必须有一个单独的结构体实例。系数数组可能可以在几个实例共享，但是变量数组不能共享。支持的 3 种数据类型都有定义不同的结构体实例类型.

初始化函数

每种数据类型都有相应的初始化函数。初始化函数处理以下操作:

- 设置数据结构字段的值
- 清零状态 buffer 的值. 不使用初始化函数，手动处理这些操作，需要设置结构体实例内的以下字段: numStages, pCoeffs, pState. 将 pState 的所有值置 0

是否使用初始化函数是可选的. 但是，如果使用初始化函数，则结构体实例不能被放在常量数据段。想要将结构体实例放在常量数据段，则必须手动的初始化结构体实例。在静态初始化之前，先把状态 buffer 中的值置 0。下列代码演示了静态初始化 3 中不同数据类型的滤波器的结构体实例。

```
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};
```

其中 numStages 是滤波器中二阶阶段的数量; pState 是状态 buffer 的地址; pCoeffs 是系数 buffer 的地址; postShift 是移位数。

定点行为

使用 Q15 和 Q31 版本的二阶级联滤波函数需要多加注意。需要考虑以下问题:

- 系数的缩放
- 滤波的增益
- 溢出和饱和

系数的缩放: 滤波器系数表示为小数值，并且系数限制在 [-1 +1) 范围之间. 定点函数有一个额外的缩放参数 postShift ，用于给滤波器系数放大到超过 [+1 -1) 范围。滤波器的累加器的输出是一个移位寄存器，结果移动 postShift 位。

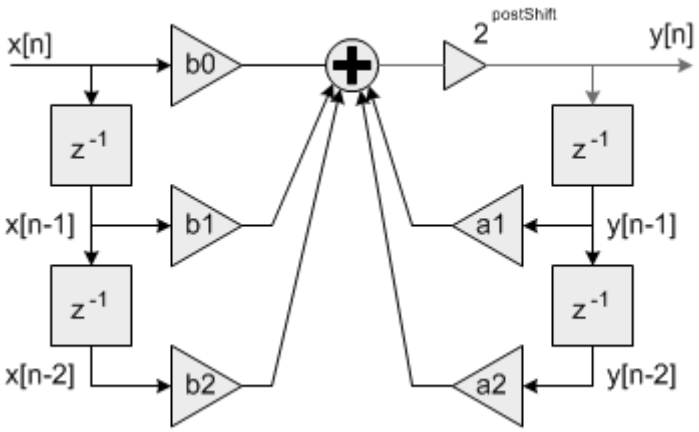


图 6.3: 定点二阶累加后带移位

这本质上将滤波器的系数缩放了 2^postShift 倍. 比如，为了表示系数

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

可以设置 pCoeffs 数组为:

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

并且设置 postShift=1

滤波器增益: 二阶滤波器的频率响应是它函数的一个系数。通过滤波器的增益可能超过 1.0，这意味着滤波器增加了某些频率的幅度。这意味着幅度小于 1.0 的输入信号可能导致输出大于 1.0，因此基于滤波器的实现，这些信号可能会饱和或溢出。为了避免这种行为，滤波器需要按比例缩小，使得其峰值增益小于 1.0，或输入信号必须按比例缩小，以便输入和滤波器的组合不会溢出

溢出和饱和: 对于 Q15 和 Q31 版本, 在各自函数的说明文档分别描述。

6.1.3 函数说明

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)
```

参数:

`*S`: 指向一个二阶级联结构实例

`*pSrc`: 指向输入数据块

`*pDst`: 指向输出数据块

`blockSize`: 每个块需要处理的样本数量

返回值:

无

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)
```

参数:

`*S`: 指向一个二阶级联结构实例

`*pSrc`: 指向输入数据块

`*pDst`: 指向输出数据块

`blockSize`: 每个块需要处理的样本数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。系数和状态变量都用 1.15 格式表示，相乘的结果是 2.30 格式。2.30 格式的中间结果用一个 64 位的累加器累加成 34.30 格式的结果。这里可以保留中间结果的精度，并且不会溢出。最后的结果，饱和转换为 1.15 格式。

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)
```

参数:

`*S`: 指向一个二阶级联结构实例

`*pSrc`: 指向输入数据块

`*pDst`: 指向输出数据块

`blockSize`: 每个块需要处理的样本数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。累加器是 2.62 格式，维护了中间结果的所有精度，但是只有一个保护位。因此，如果累加器结果溢出，会往符号位溢出，而不是截断。为了防止溢出，输入信号必须缩小 2 位，缩放到 [-0.25 +0.25) 区间. 在处理 5 个相乘结果累加之后，2.62 累加器移动 postShift 位，并且丢弃低 32 位，将结果截断为 1.31 格式。

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)
```

参数:

`*S`: 指向一个二阶级联结构实例

`*pSrc`: 指向输入数据块

`*pDst`: 指向输出数据块

`blockSize`: 每个块需要处理的样本数量

返回值:

无

缩放和溢出行为:

快速版本使用一个 2.30 格式的 32 位累加器。累加器维持了中间乘法结果的所有精度，但是只有一个守护位。因此，如果累加器结果溢出，会往符号位覆盖，而不是截断。为了防止溢出，输入信号必须先缩小两个位，到 [-0.25 +0.25) 范围。2.30 累加器的结果移动 postShift 位，并且结果通过丢弃低 16 位，截断为 1.15 格式

参考函数*csi_biquad_cascade_df1_q15()* 使用了 64 位累加器实现了一个更慢的版本防止精度丢失。慢的和快的版本使用了相同的结构体实例。使用函数*csi_biquad_cascade_df1_q15()* 初始化滤波器结构.

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)
```

参数:

`*S`: 指向一个二阶级联结构实例

`*pSrc`: 指向输入数据块

`*pDst`: 指向输出数据块

`blockSize`: 每个块需要处理的样本数量

返回值:

无

缩放和溢出行为:

该函数针对速度进行了优化，牺牲了定点精度和溢出保护. 每个 1.31 和 1.31 相乘的结果截断为 2.30 格式。这些中间结果相加到一个 2.30 累加器。最后，累加器饱和并且转换为 1.31 的结果。快速版本具有与标准版本相同的溢出行为，但是提供较低的精度，因为它丢弃每个乘法结果的低 32 位。为了防止溢出，输入信号必须先缩小两个位，到 [-0.25 +0.25) 范围。使用初始化函数*csi_biquad_cascade_df1_init_q31()* 来初始化滤波器结构体.

参数函数*csi_biquad_cascade_df1_q31()* 是这个函数的一个更慢的实现，为更高精度使用了一个 64 位累加器。慢速和快速的版本都使用了相同的结构体。使用函数*csi_biquad_cascade_df1_init_q31()* 来初始化滤波器结构体.

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)
```

参数:

`*S`: 指向二阶级联滤波器结构体实例

`numStages`: 滤波器中二阶阶段的数量

`*pCoeffs`: 指向滤波器系数数组

`*pState`: 指向状态数组

返回值:

无

系数和状态顺序:

系数按以下顺序保存在数组 pCoeffs :

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

其中 b1x 和 a1x 是第一个阶段的系数, b2x 和 a2x 是第二个阶段的系数, 依次类推。pCoeffs 数组总共有 5*numStages 个数值。

pState 是指向状态数组的指针. 每个二阶阶段有 4 个状态变量 x[n-1], x[n-2], y[n-1], 和 y[n-2]. 状态变量排列在 pState 数组的顺序如下:

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

最前面是第一阶段的 4 个状态变量, 然后是第二阶段的 4 个状态变量, 依次类推。状态数组总共有 4*numStages 个数值. 状态变量在数据块处理之后更新，系数不会更新。

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)
```

参数:

`*S`: 指向二阶级联滤波器结构体实例

`numStages`: 滤波器中二阶阶段的数量

`*pCoeffs`: 指向滤波器系数数组

`*pState`: 指向状态数组

`postShift` 累加器结果的移位数

返回值:

无

系数和状态顺序:

系数按以下顺序保存在数组 pCoeffs :

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

其中 b1x 和 a1x 是第一阶段的系数, b2x 和 a2x 是第二阶段的系数, 依次类推。pCoeffs 有 6*numStages 个值. b1 和 b2 中间的填充的 16 位的 0，是为了让系数能够 32 位对齐。

状态变量保存在数组 pState. 每个二阶阶段有 4 个状态变量 x[n-1], x[n-2], y[n-1], 和 y[n-2]. 状态变量排列在 pState 数组中的形式如下:

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

阶段 1 使用最开始的 4 个状态变量，阶段 2 使用接下来的 4 个状态变量，依次类推。状态数组总共有 4*numStages 个值. 每个块数据处理会更新状态变量，不会更新系数。

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)
```

参数:

*S: 指向二阶级联滤波器结构体实例

numStages: 滤波器中二阶阶段的数量

*pCoeffs: 指向滤波器系数数组

*pState: 指向状态数组

postShift: 累加器结果的移位数

返回值:

无

Coefficient and State Ordering:

保存在数组 pCoeffs 的系数，按照以下顺序排列:

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

其中 b1x 和 a1x 是第一阶段的系数, b2x 和 a2x 是第二阶段的系数, 依次类推。pCoeffs 数组总共有 5*numStages 个值。

pState 指向状态变量数组。每个二阶阶段有 4 个状态变量 x[n-1], x[n-2], y[n-1], 和 y[n-2]. 状态变量在数组 pState 中的排列顺序:

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

阶段 1 使用最开始的 4 个状态变量，阶段 2 使用接下来的 4 个状态变量，依次类推。状态变量数组总共有 4*numStages 个值. 每个块数据处理会更新状态变量，不会更新系数。

6.2　直接 II 型 IIR 滤波器

6.2.1　函数

- *csi_biquad_cascade_df2T_f32* : 浮点转置直接 II 型二阶级联滤波器处理函数
- *csi_biquad_cascade_df2T_init_f32* : 浮点转置直接 II 型二阶级联滤波器初始化函数
- *csi_biquad_cascade_stereo_df2T_f32* : 浮点转置直接 II 型二阶级联滤波器处理函数
- *csi_biquad_cascade_stereo_df2T_init_f32* : 浮点转置直接 II 型二阶级联滤波器初始化函数

6.2.2　简要说明

这些函数使用了直接 II 型转置结构实现了任意阶递归（IIR）滤波器。滤波器是由二阶单元 Biquad 级联实现。

与直接 I 型相比，这些函数稍微节省了一些内存占用。

只支持浮点数据类型。

函数以块单位操作输入输出数据。每次调用函数通过滤波器处理 blockSize 个样本。pSrc 指向输入数组数据，pDst 指向输出数组数据。两个数组都包含 blockSize 个数值。

算法

每个二阶阶段都使用差分方程实现一个二阶滤波：

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

其中 d1 和 d2 表示两个状态变量.

使用单个转置直接 II 型结构的双二阶滤波器如下：

系数 b0, b1, 和 b2 与输入信号 x[n] 相乘，并且被称为前馈系数。系数 a1 和 a2 与输出信号 y[n] 相乘，并且被称为反馈系数。需要注意反馈系数的符号。有些设计工具使用差分方程：

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

这种情况下，反馈系数 a1 和 a2 在使用 CSI DSP 库的时候，必须取反。

高阶滤波器通过级联的二阶单元实现。numStages 指定使用了多少二阶阶段。比如，第 8 阶滤波器是 numStages=4 的二阶阶段。第 9 阶滤波器可以看做 numStages=5 的二阶阶段，系数配置为第一阶 (b2=0 和 a2=0)。

pState 指向状态变量数组。每个二阶阶段有 2 个状态变量 d1 和 d2. 状态变量被排列在 pState 如下:

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

其中 d1x 是第一个二阶阶段的状态变量，d2x 是第二个二阶阶段的变量，状态变量数组总共有 2*numStages 个值. 每个块数据处理会更新状态变量，不会更新系数。

CSI 库里面包括了直接 I 型和转置直接 II 型。直接 I 型的优点是，对于定点数据类型会更加稳定。这也是为什么直接 I 型提供了 Q15 和 Q31 数据类型的函数。另一方面，转置直接 II 型结构，需要更广的动态范围状

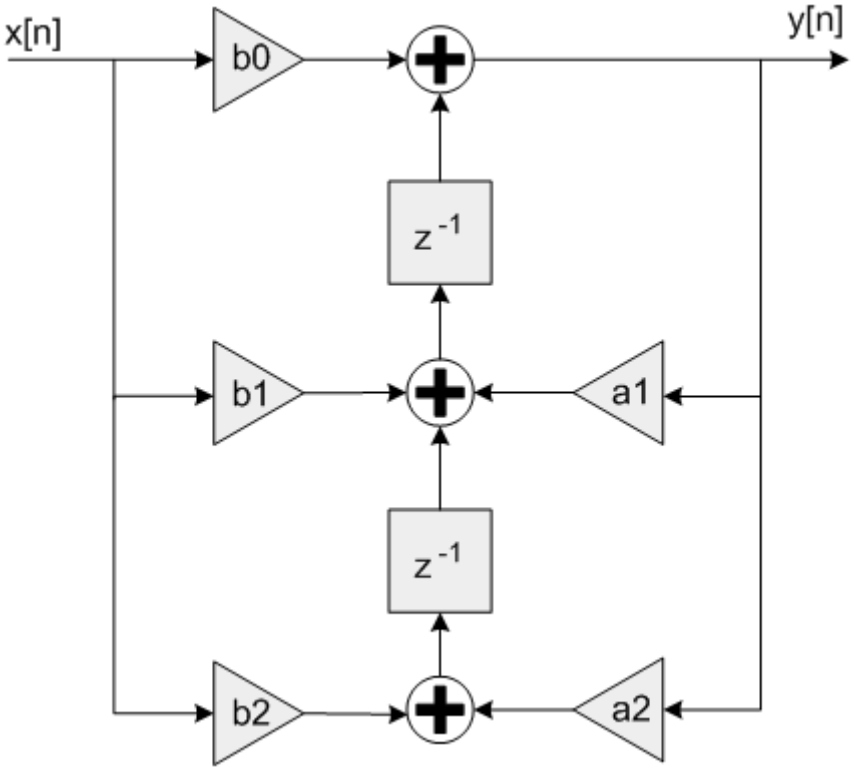


图 6.4: 单转置直接 II 型

态变量 d1 和 d2. 因此，CSI 库只提供浮点版本的直接 II 型函数。直接 II 型的优点是，每个阶段只需要 2 个状态变量。

结构体实例

滤波器的系数和状态变量都保存在数据结构体实例中。每个滤波器都需要定义单独的结构体实例。系数数组可以在几个实例中共享，但是状态变量数组不能共享。

初始化函数

提供一个对应的初始化函数。初始化函数处理下列操作：

- 设置结构体内字段的值
- 清零状态 buffer 的值不使用初始化函数，手动处理这些操作，需要设置结构体实例内的以下字段：numStages, pCoeffs, pState. 将 pState 的所有值置 0。

是否使用初始化函数是可选的。但是，如果使用初始化函数，则结构体实例不能被放在常量数据段。想要将结构体实例放在常量数据段，则必须手动的初始化结构体实例。在静态初始化之前，先把状态 buffer 中的值置 0。比如，静态初始化结构体实例使用：

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

其中 numStages 是滤波器内二阶阶段的数量; pState 是状态 buffer 的地址。pCoeffs 是系数 buffer 的地址。

6.2.3 函数说明

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)
```

参数:

`*S`: 指向滤波器数据结构体实例

`*pSrc`: 指向输入数据

`*pDst`: 指向输出数据

`blockSize`: 处理的样本数量

返回值:

无

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)
```

参数:

*S: 指向滤波器数据结构体实例

numStages: 滤波器中二阶阶段的数量

*pCoeffs: 指向滤波器系数

*pState: 指向状态数组

返回值:

无

系数和状态顺序:

保存在数组 pCoeffs 中的系数，按下列顺序排列:

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

其中 b1x 和 a1x 是第一阶段的系数, b2x 和 a2x 是第二阶段的系数, 依次类推。pCoeffs 数组总共有 5*numStages 个数。

pState 是指向状态数组的指针。每个二阶阶段有 2 个状态变量 d1, 和 d2. 阶段 1 使用最开始的 2 个状态变量，阶段 2 使用接下来的 2 个状态变量，依次类推。状态数组总共有 2*numStages 个值. 每个块数据处理会更新状态变量，不会更新系数。

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)
```

参数:

*S: 指向滤波器数据结构体实例

*pSrc: 指向输入数据

*pDst: 指向输出数据

blockSize: 处理的样本数量

返回值:

无

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)
```

参数:

*S: 指向滤波器数据结构体实例

numStages: 滤波器中二阶阶段的数量

*pCoeffs: 指向滤波器系数

*pState: 指向状态数组

返回值:

无

系数和状态顺序:

保存在数组 pCoeffs 中的系数，按下列顺序排列:

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

其中 b1x 和 a1x 是第一阶段的系数, b2x 和 a2x 是第二阶段的系数, 依次类推。pCoeffs 数组总共有 5*numStages 个数值。

pState 是指向状态数组的指针。每个二阶阶段有 2 个状态变量 d1, 和 d2 . 阶段 1 使用最开始的 2 个状态变量，阶段 2 使用接下来的 2 个状态变量，依次类推。状态数组总共有 2*numStages 个值. 每个块数据处理会更新状态变量，不会更新系数。

6.3 卷积

6.3.1 函数

- *csi_conv_f32* : 浮点序列的卷积
- *csi_conv_q31* : Q31 序列的卷积
- *csi_conv_q15* : Q15 序列的卷积
- *csi_conv_q7* : Q7 序列的卷积
- *csi_conv_fast_q31* : Q31 序列的卷积
- *csi_conv_fast_q15* : Q15 序列的卷积
- *csi_conv_fast_opt_q15* : Q15 序列的卷积
- *csi_conv_opt_q15* : Q15 序列的卷积
- *csi_conv_opt_q7* : Q7 序列的卷积

6.3.2 简要说明

卷积是一种控制两个有限向量生成一个有限向量的数学操作。卷积类似于相关分析，经常用在过滤和数据分析。CSI DSP 库包括了 Q7，Q15，Q31 和浮点数据类型的卷积函数。

算法

令 a[n] 和 b[n] 分别是长度 srcALen 和 srcBLen 的样本序列. 则卷积是

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

如下定义

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

注意：c[n] 的长度是 srcALen + srcBLen - 1 ，并且在区间 n=0, 1, 2, …, srcALen + srcBLen - 2 内. pSrcA 指向第一个输入长度向量 srcALen ，pSrcB 指向第二个输入长度向量 srcBLen. 输出结果写入到 pDst ，调用函数必须为结果分配 srcALen+srcBLen-1 个字的空间。

概念上，当两个信号 a[n] 和 b[n] 卷积时，信号 b[n] 在 a[n] 上滑动。对于每个偏移 n ，a[n] 和 b[n] 的重叠部分被相乘并相加在一起。

注意，卷积是可交换的操作:

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

这意味着交换 A 和 B 参数不会对卷积函数造成影响。

定点行为

卷积操作会产生大量的中间结果。因此，Q7，Q15 和 Q31 函数会有溢出和饱和的风险。参考每个函数各自的文档说明，使用的算法的细节情况。

快速版本

Q31 和 Q15 支持快速版本。快速版本需要的周期数更少，但是需要缩放输入信号，确保不会引起中间值溢出

Opt 版本

Q15 和 Q7 有 Opt 版本. 设计使用内部 buffer 来达到更好的优化效果。这些版本优化了速度，但是消耗更多的内存。

6.3.3 函数说明

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)
```

参数:

*pSrcA: 指向第一个输入序列

srcALen: 第一个输入序列的长度

*pSrcB: 指向第二个输入序列

srcBLen: 第二个输入序列的长度

*pDst: 指向输出结果的地址，长度是 srcALen+srcBLen-1

返回值:

无

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)
```

参数:

*pSrcA: 指向第一个输入序列

srcALen: 第一个输入序列的长度

*pSrcB: 指向第二个输入序列

srcBLen: 第二个输入序列的长度

*pDst: 指向输出结果的地址，长度是 srcALen+srcBLen-1

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入都表示为 1.15 格式，相乘的结果是 2.30 格式。2.30 的中间结果在 34.30 格式的 64 位累加器中累加。由于有 33 个守护位，所以应该不会有溢出 34.30 格式的结果丢弃低 15 位，截断为 34.15 格式，然后再饱和为 1.15 格式。

函数*csi_conv_fast_q15()* 是一个快速版本，但是丢失了更多的精度。

函数*csi_conv_opt_q15()* 是一个快速版本，使用了额外的临时缓存空间。

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)
```

参数:

`*pSrcA`: 指向第一个输入序列

`srcALen`: 第一个输入序列的长度

`*pSrcB`: 指向第二个输入序列

`srcBLen`: 第二个输入序列的长度

`*pDst`: 指向输出结果的地址，长度是 srcALen+srcBLen-1

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。累加器使用 2.62 格式维持所有的中间乘法结果的精度，但是只有一个保护位累加过程中没有饱和操作。如果累加器溢出，会往符号位溢出，扭曲结果。因此，输入信号需要缩小来防止溢出。因为加法过程中最多会有 min(srcALen, srcBLen) 次进位，所以，输入需要缩小 log2(min(srcALen, srcBLen)) (log2 是 2 为底的对数) 倍来防止溢出。2.62 格式的累加器，右移 31 位，再饱和生成 1.31 格式的最后结果。

csi_conv_fast_q31() 是一个快速版本，但是丢失了更多精度。

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)
```

参数:

`*pSrcA`: 指向第一个输入序列

`srcALen`: 第一个输入序列的长度

`*pSrcB`: 指向第二个输入序列

`srcBLen`: 第二个输入序列的长度

`*pDst`: 指向输出结果的地址，长度是 srcALen+srcBLen-1

返回值:

无

缩放和溢出行为:

函数使用了一个内部 32 位累加器。输入都用 1.7 格式表示，相乘的结果是 2.14 格式。2.14 的中间结果在 18.14 格式的 32 位累加器中累加。由于有 17 个守护位，除非 max(srcALen, srcBLen) 大于 131072，否则不会溢出。18.14 格式的结果丢弃低 7 位，截断为 18.7 格式，然后再饱和成 1.7 格式。

函数*csi_conv_opt_q7()* 是这个函数的一个快速版本。

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)
```

参数:

`*pSrcA`: 指向第一个输入序列

`srcALen`: 第一个输入序列的长度

`*pSrcB`: 指向第二个输入序列

`srcBLen`: 第二个输入序列的长度

`*pDst`: 指向输出结果的地址，长度是 srcALen+srcBLen-1

`*pScratch1` 指向临时 buffer，大小是 max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2

`*pScratch2` 指向临时 buffer，大小是 min(srcALen, srcBLen)

返回值:

无

限制:

如果芯片不支持分对齐访问，输入，输出，临时 buffer，都应该是 32 位对齐

缩放和溢出行为:

这个快速版本使用一个格式为 2.30 的 32 位累加器。累加器维持了中间相乘结果的所有精度，但是只有一个保护位。因为中间加法没有饱和操作，所以，累加器如果溢出的话，会往符号位溢出，扭曲结果。

输入信号需要缩放到防止中间结果溢出。因为，最多可能有 min(srcALen, srcBLen) 个内部加法进位，所以输入需要缩放 log2(min(srcALen, srcBLen)) (log2 是 2 为底的对数) 倍。2.30 格式的累加器右移 15 位，并且饱和到 1.15 格式生成最后的结果。

函数*csi_conv_q15()* 是一个慢速实现，使用了 64 位累加器，来防止精度丢失。

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)
```

参数:

*pSrcA: 指向第一个输入序列

srcALen: 第一个输入序列的长度

*pSrcB: 指向第二个输入序列

srcBLen: 第二个输入序列的长度

*pDst: 指向输出结果的地址，长度是 srcALen+srcBLen-1

返回值:

无

缩放和溢出行为:

这个快速版本使用一个格式为 2.30 的 32 位累加器。累加器维持了中间相乘结果的所有精度，但是只有一个保护位。因为中间加法没有饱和操作，所以，累加器如果溢出的话，会往符号位溢出，扭曲结果。

输入信号需要缩放到防止中间结果溢出。因为，最多可能有 min(srcALen, srcBLen) 个内部加法进位，所以输入需要缩放 log2(min(srcALen, srcBLen)) (log2 是 2 为底的对数) 倍。2.30 格式的累加器右移 15 位，并且饱和到 1.15 格式生成最后的结果。

函数*csi_conv_q15()* 是一个慢速实现，使用了 64 位累加器，来防止精度丢失。

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)
```

参数:

*pSrcA: 指向第一个输入序列

srcALen: 第一个输入序列的长度

*pSrcB: 指向第二个输入序列

srcBLen: 第二个输入序列的长度

*pDst: 指向输出结果的地址，长度是 srcALen+srcBLen-1

返回值:

无

缩放和溢出行为:

该函数针对速度进行了优化，牺牲了定点精度和溢出保护. 每个 1.31 和 1.31 相乘的结果截断为 2.30 格式. 这些中间结果以 2.30 格式在一个 32 位寄存器中累加. 最后，累加器饱和转换为 1.31 结果.

快速版本具有与标准版本相同的溢出行为，但提供较小的精度，因为它丢弃每个乘法结果的低 32 位为了避免溢出，输入信号必须缩放。因为，最多可能有 min(srcALen, srcBLen) 个内部加法进位，所以输入需要缩放 log2(min(srcALen, srcBLen)) (log2 是 2 为底的对数) 倍。

见*csi_conv_q31()*，这个函数是一个慢速实现，使用了 64 位累加器，来防止精度丢失。

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)
```

参数:

`*pSrcA`: 指向第一个输入序列

`srcALen`: 第一个输入序列的长度

`*pSrcB`: 指向第二个输入序列

`srcBLen`: 第二个输入序列的长度

`*pDst`: 指向输出结果的地址，长度是 srcALen+srcBLen-1

`*pScratch1` 指向临时 buffer，大小是 max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2

`*pScratch2` 指向临时 buffer，大小是 min(srcALen, srcBLen)

返回值:

无

限制:

如果芯片不支持分对齐访问，输入，输出，临时 buffer，都应该是 32 位对齐。

缩放和溢出行为:

函数实现使用了一个内部 64 位内部累加器。输入都是 1.15 格式，相乘结果是 2.30 格式。2.30 的中间结果在 64 位累加器中以 34.30 格式保存。这种方法提供了 33 个保护位，因此没有溢出的风险。最后，34.30 格式的结果丢弃低 15 位截断为 34.15 格式，然后饱和成 1.15 格式的结果。

参考*csi_conv_fast_q15()*，是一个快速，但是降低了精度的实现版本。

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)
```

参数:

`*pSrcA`: 指向第一个输入序列

`srcALen`: 第一个输入序列的长度

`*pSrcB`: 指向第二个输入序列

srcBLen：第二个输入序列的长度

*pDst：指向输出结果的地址，长度是 srcALen+srcBLen-1

*pScratch1 指向临时 buffer，大小是 max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2

*pScratch2 指向临时 buffer，大小是 min(srcALen, srcBLen)

返回值:

无

限制:

如果芯片不支持分对齐访问，输入，输出，临时 buffer，都应该是 32 位对齐。

缩放和溢出行为:

函数实现使用了一个内部 32 位累加器。输入都表示为 1.7 格式，相乘的结果是 2.14 格式。2.14 的中间结果在 18.14 格式的 32 位累加器中累加。由于有 17 个守护位，除非 max(srcALen, srcBLen) 大于 131072，否则不会溢出。18.14 格式的结果丢弃低 7 位，截断为 18.7 格式，然后再饱和成 1.7 格式。

6.4 部分卷积

6.4.1 函数

- *csi_conv_partial_f32* : 浮点序列的部分卷积
- *csi_conv_partial_q31* : Q31 序列的部分卷积
- *csi_conv_partial_q15* : Q15 序列的部分卷积
- *csi_conv_partial_q7* : Q7 序列的部分卷积
- *csi_conv_partial_fast_q31* : Q31 序列的部分卷积
- *csi_conv_partial_fast_q15* : Q15 序列的部分卷积
- *csi_conv_partial_fast_opt_q15* : Q15 序列的部分卷积
- *csi_conv_partial_opt_q15* : Q15 序列的部分卷积
- *csi_conv_partial_opt_q7* : Q7 序列的部分卷积

6.4.2 简要说明

部分卷积等价于卷积，只是生成的输出样本是卷积的子集。每个函数有两个额外添加的参数。firstIndex 指定输出样本子集的开始序号。numPoints 是需要计算的输出样本的数量函数计算的输出范围在：[firstIndex, ···, firstIndex+numPoints-1]. 输出数组 pDst 包括 numPoints 个值.

可选的输出范围在 [0 srcALen+srcBLen-2]. 如果请求的自己并不在这个范围，则函数返回 CSKY_MATH_ARGUMENT_ERROR. 否则函数返回 CSKY_MATH_SUCCESS.

定点行为

定点的行为参考卷积。

快速版本

Q31 和 Q15 的部分卷积有快速版本。快速版本需要的周期更多，但是，设计为需要输入信号缩放到不会引起中间计算发生溢出。

Opt 版本

Q15 和 Q7 有 Opt 版本. 设计使用内部 buffer 来达到更好的优化效果。这些版本优化了速度，但是相对的消耗更多的内存。

6.4.3 函数说明

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)
```

参数:

*pSrcA: 指向第一个输入序列

srcALen: 第一个输入序列的长度

*pSrcB: 指向第二个输入序列

srcBLen: 第二个输入序列的长度

*pDst: 指向输出结果的地址

firstIndex: 输出样本的起始索引

numPoints: 输出样本的数量

返回值:

函数正确完成返回 CSKY_MATH_SUCCESS ，如果请求的子集超出范围 [0 srcALen+srcBLen-2]，则返回 CSKY_MATH_ARGUMENT_ERROR 。

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)
```

参数:

*pSrcA: 指向第一个输入序列

srcALen: 第一个输入序列的长度

*pSrcB: 指向第二个输入序列

srcBLen: 第二个输入序列的长度

*pDst: 指向输出结果的地址

firstIndex: 输出样本的起始索引

numPoints: 输出样本的数量

返回值:

函数正确完成返回 CSKY_MATH_SUCCESS ，如果请求的子集超出范围 [0 srcALen+srcBLen-2]，则返回 CSKY_MATH_ARGUMENT_ERROR。

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)
```

参数:

*pSrcA: 指向第一个输入序列

srcALen: 第一个输入序列的长度

*pSrcB: 指向第二个输入序列

srcBLen: 第二个输入序列的长度

*pDst: 指向输出结果的地址
firstIndex: 输出样本的起始索引
numPoints: 输出样本的数量

返回值:

函数正确完成返回 CSKY_MATH_SUCCESS ，如果请求的子集超出范围 [0 srcALen+srcBLen-2]，则返回 CSKY_MATH_ARGUMENT_ERROR

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)
```

参数:

*pSrcA: 指向第一个输入序列
srcALen: 第一个输入序列的长度
*pSrcB: 指向第二个输入序列
srcBLen: 第二个输入序列的长度
*pDst: 指向输出结果的地址
firstIndex: 输出样本的起始索引
numPoints: 输出样本的数量

返回值:

函数正确完成返回 CSKY_MATH_SUCCESS ，如果请求的子集超出范围 [0 srcALen+srcBLen-2]，则返回 CSKY_MATH_ARGUMENT_ERROR

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)
```

参数:

*pSrcA: 指向第一个输入序列
srcALen: 第一个输入序列的长度
*pSrcB: 指向第二个输入序列
srcBLen: 第二个输入序列的长度
*pDst: 指向输出结果的地址
firstIndex: 输出样本的起始索引
numPoints: 输出样本的数量
*pScratch1 指向缓存地址，大小为 max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2
*pScratch2 指向缓存地址，大小为 min(srcALen, srcBLen)

返回值:

函数正确完成返回 CSKY_MATH_SUCCESS ，如果请求的子集超出范围 [0 srcALen+srcBLen-2]，则返回 CSKY_MATH_ARGUMENT_ERROR

限制:

如果芯片不支持分对齐访问，输入，输出，临时 buffer，都应该是 32 位对齐。

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)
```

参数:

*pSrcA: 指向第一个输入序列
srcALen: 第一个输入序列的长度
*pSrcB: 指向第二个输入序列
srcBLen: 第二个输入序列的长度
*pDst: 指向输出结果的地址
firstIndex: 输出样本的起始索引
numPoints: 输出样本的数量

返回值:

函数正确完成返回 CSKY_MATH_SUCCESS ，如果请求的子集超出范围 [0 srcALen+srcBLen-2]，则返回 CSKY_MATH_ARGUMENT_ERROR

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)
```

参数:

*pSrcA: 指向第一个输入序列
srcALen: 第一个输入序列的长度
*pSrcB: 指向第二个输入序列
srcBLen: 第二个输入序列的长度
*pDst: 指向输出结果的地址
firstIndex: 输出样本的起始索引
numPoints: 输出样本的数量

返回值:

函数正确完成返回 CSKY_MATH_SUCCESS ，如果请求的子集超出范围 [0 srcALen+srcBLen-2]，则返回 CSKY_MATH_ARGUMENT_ERROR

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)
```

参数:

*pSrcA: 指向第一个输入序列

srcALen: 第一个输入序列的长度

*pSrcB: 指向第二个输入序列

srcBLen: 第二个输入序列的长度.

*pDst: 指向输出结果的地址

firstIndex: 输出样本的起始索引

numPoints: 输出样本的数量

*pScratch1 指向临时缓存，大小为 max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2

*pScratch2 指向临时缓存，大小为 min(srcALen, srcBLen)

返回值:

函数正确完成返回 CSKY_MATH_SUCCESS ，如果请求的子集超出范围 [0 srcALen+srcBLen-2]，则返回 CSKY_MATH_ARGUMENT_ERROR

限制:

如果芯片不支持分对齐访问，输入，输出，临时 buffer，都应该是 32 位对齐。

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)
```

参数:

*pSrcA: 指向第一个输入序列

srcALen: 第一个输入序列的长度

*pSrcB: 指向第二个输入序列

srcBLen: 第二个输入序列的长度.

*pDst: 指向输出结果的地址

firstIndex: 输出样本的起始索引

numPoints: 输出样本的数量

*pScratch1 指向临时缓存，大小为 max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2

*pScratch2 指向临时缓存，大小为 min(srcALen, srcBLen)

返回值:

函数正确完成返回 CSKY_MATH_SUCCESS ，如果请求的子集超出范围 [0 srcALen+srcBLen-2]，则返回 CSKY_MATH_ARGUMENT_ERROR

限制:

如果芯片不支持分对齐访问，输入，输出，临时 buffer，都应该是 32 位对齐。

6.5 相关分析

6.5.1 函数

- *csi_correlate_f32* : 浮点序列的相关
- *csi_correlate_q31* : Q31 序列的相关
- *csi_correlate_q15* : Q15 序列的相关
- *csi_correlate_q7* : Q7 序列的相关
- *csi_correlate_fast_q31* : Q31 序列的相关
- *csi_correlate_fast_q15* : Q15 序列的相关
- *csi_correlate_fast_opt_q15* : Q15 序列的相关
- *csi_correlate_opt_q15* : Q15 序列的相关
- *csi_correlate_opt_q7* : Q7 序列的相关

6.5.2 简要说明

相关是一种与卷积类似的数学操作。跟卷积一样，相关用两个信号生成一个新的信号。相关和卷积中的基本算法是相同的，除了其中一个输入在卷积中被翻转。相关一般用来测量两个信号的相似性。广泛应用在模式识别，密码分析和搜索。CSI 库为 Q7, Q15, Q31 和浮点数据类型提供相关函数。Q15 和 Q31 还提供了快速版本函数。

算法

令 a[n] 和 b[n] 分别是长度为 srcALen 和 srcBLen 样本序列. 两个信号的卷积表示为:

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

相关则让其中一个信号翻转:

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

下面是数学定义

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

pSrcA 指向第一个输入序列，序列长度是 srcALen ， pSrcB 指向第二个输入序列，序列长度是 srcBLen. 结果 c[n] 的长度是 2 * max(srcALen, srcBLen) - 1 ， 所有结果都落在区间 n=0, 1, 2, ⋯, (2 * max(srcALen, srcBLen) - 2) 中. 输出结果写入到 pDst ， 调用函数必须分配好 2 * max(srcALen, srcBLen) - 1 个字保存结果。

注意

pDst 在使用前需要先清零。

定点行为

相关会生成大量的中间值。因此，Q7, Q15, 和 Q31 函数有可能会发生溢出和饱和。参考具体函数的文档，了解使用特定算法的详细情况。

快速版本

Q31 和 Q15 支持快速版本。快速版本需要更少的周期数，但是设计为，需要输入信号缩小到不会引起中间结果溢出。

Opt 版本

Q15 和 Q7 支持 Opt 版本。设计使用临时缓存来获取更好的优化效果。这些版本优化了速度，但是相对的消耗了更多的内存 (临时缓存)。

6.5.3 函数说明

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)
```

参数:

`*pSrcA`: 指向第一个输入序列

`srcALen`: 第一个输入序列的长度

`*pSrcB`: 指向第二个输入序列

`srcBLen`: 第二个输入序列的长度

`*pDst`: 指向输出结果的地址，长度是 2 * max(srcALen, srcBLen) - 1

返回值:

无

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)
```

参数:

`*pSrcA`: 指向第一个输入序列

`srcALen`: 第一个输入序列的长度

`*pSrcB`: 指向第二个输入序列

`srcBLen`: 第二个输入序列的长度

`*pDst`: 指向输出结果的地址，长度是 2 * max(srcALen, srcBLen) - 1

返回值:

无

缩放和溢出行为:

函数实现使用了一个 64 位内部累加器。累加器使用 2.62 格式，维持了中间乘法结果的所有精度，但是只有一个保护位。累加过程中没有饱和操作。因此，如果累加器溢出，会往符号位溢出，扭曲结果。输入信号需要缩小来防止中间结果溢出。因为加法最多发生 min(srcALen, srcBLen) 次进位，所以需要缩小输入 1/min(srcALen, srcBLen) 来防止溢出。2.62 累加器右移 31 位，然后饱和生成为 1.31 格式的结果。

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)
```

参数:

`*pSrcA`: 指向第一个输入序列

`srcALen`: 第一个输入序列的长度

`*pSrcB`: 指向第二个输入序列

`srcBLen`: 第二个输入序列的长度

`*pDst`: 指向输出结果的地址，长度是 2 * max(srcALen, srcBLen) - 1

返回值:

无

缩放和溢出行为:

函数实现使用了一个 64 位内部累加器。输入都是 1.15 格式，相乘的结果是 2.30 格式。2.30 格式的结果在 34.30 格式的 64 位累加器中累加。由于有 33 个守护位，不会有溢出风险。34.30 格式的结果丢弃低 15 位，截断为 34.15 格式，然后再饱和成 1.15 格式。

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)
```

参数:

`*pSrcA`: 指向第一个输入序列

`srcALen`: 第一个输入序列的长度

`*pSrcB`: 指向第二个输入序列

`srcBLen`: 第二个输入序列的长度

`*pDst`: 指向输出结果的地址，长度是 2 * max(srcALen, srcBLen) - 1

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 32 位累加器。输入都是 1.7 格式，相乘的结果是 2.14 格式。2.14 格式的结果在 18.14 格式的 32 位累加器中累加。由于有 17 个保护位，除非 max(srcALen, srcBLen) 大于 131072，否则不会溢出 18.14 的结果丢弃低 7 位，截断为 18.7 格式，最后饱和为 1.7 格式。

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)
```

参数:

`*pSrcA`: 指向第一个输入序列

`srcALen`: 第一个输入序列的长度

`*pSrcB`: 指向第二个输入序列

`srcBLen`: 第二个输入序列的长度

`*pDst`: 指向输出结果的地址，长度是 2 * max(srcALen, srcBLen) - 1

`*pScratch` 指向临时缓存，大小是 max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.

返回值:

无

限制:

如果芯片不支持分对齐访问，输入，输出，临时 buffer，都应该是 32 位对齐。

缩放和溢出行为:

这个快速版本使用了一个 2.30 格式的 32 位累加器。累加器维持了中间乘法结果的全部精度，但是只有 1 个保护位。累加过程中没有饱和操作。因此，如果累加器溢出，会扭曲结果。为了防止中间结果溢出，必须缩小输入信号。因为加法最多会有 min(srcALen, srcBLen) 个进位，所以，输入信号需要缩小 1/min(srcALen, srcBLen) 防止溢出。2.30 格式累加器右移 15 位，然后饱和生成 1.15 的最后结果。

函数*csi_correlate_q15()* 是这个函数的一个慢速版本，使用了一个 64 位累加器来防止溢出。

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)
```

参数:

*pSrcA: 指向第一个输入序列

srcALen: 第一个输入序列的长度

*pSrcB: 指向第二个输入序列

srcBLen: 第二个输入序列的长度

*pDst: 指向输出结果的地址，长度是 2 * max(srcALen, srcBLen) - 1

返回值:

无

缩放和溢出行为:

这个快速版本使用一个 2.30 格式的 32 位累加器。累加器维持了中间乘法结果的全部精度，但是只有 1 个保护位。累加过程中没有饱和操作。因此，如果累加器溢出，会扭曲结果。为了防止中间结果溢出，必须缩小输入信号。因为加法最多会有 min(srcALen, srcBLen) 个进位，所以，输入信号需要缩小 1/min(srcALen, srcBLen) 防止溢出。2.30 格式累加器右移 15 位，然后饱和生成 1.15 的最后结果。

函数*csi_correlate_q15()* 是这个函数的一个慢速版本，使用了一个 64 位累加器来防止溢出。

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)
```

参数:

*pSrcA: 指向第一个输入序列

srcALen: 第一个输入序列的长度

*pSrcB: 指向第二个输入序列

srcBLen: 第二个输入序列的长度

*pDst: 指向输出结果的地址，长度是 2 * max(srcALen, srcBLen) - 1

返回值:

无

缩放和溢出行为:

该函数针对速度进行了优化，牺牲了定点精度和溢出保护。每个 1.31 和 1.31 乘法的结果截断为 2.30 格式. 中间结果在 32 位寄存器中累加为 2.30 格式的结果。最后，累加器饱和转换为 1.31 的结果。

快速版本和标准版本的溢出行为一样，不过，丢弃低 32 位，导致精度会更低。为了避免中间结果溢出，输入信号必须缩放。因为加法最多发生 min(srcALen, srcBLen) 次进位，所以需要缩小输入 1/min(srcALen, srcBLen) 来防止溢出。

函数*csi_correlate_q31()* 是这个函数的一个慢速版本，使用了 64 位累加器提供更高的精度。

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)
```

参数:

*pSrcA: 指向第一个输入序列

srcALen: 第一个输入序列的长度

*pSrcB: 指向第二个输入序列

srcBLen: 第二个输入序列的长度

*pDst: 指向输出结果的地址，长度是 2 * max(srcALen, srcBLen) - 1

*pScratch 指向临时缓存，大小是 max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.

返回值:

无

限制:

如果芯片不支持分对齐访问，输入，输出，临时 buffer，都应该是 32 位对齐。

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入都表示为 1.15 格式，相乘的结果是 2.30 格式。2.30 的中间结果在 34.30 格式的 64 位累加器中累加。由于有 33 个守护位，不会有溢出风险。34.30 格式的结果丢弃低 15 位，截断为 34.15 格式，然后再饱和成 1.15 格式。

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)
```

参数:

*pSrcA: 指向第一个输入序列

srcALen: 第一个输入序列的长度

*pSrcB: 指向第二个输入序列

srcBLen: 第二个输入序列的长度

*pDst: 指向输出结果的地址，长度是 2 * max(srcALen, srcBLen) - 1

*pScratch1 指向临时缓存，大小是 max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.

*pScratch2 指向临时 buffer，大小是 min(srcALen, srcBLen).

返回值:

无

限制:

如果芯片不支持分对齐访问，输入，输出，临时 buffer，都应该是 32 位对齐。

缩放和溢出行为:

函数实现使用了一个内部 32 位累加器。输入都表示为 1.7 格式，相乘的结果是 2.14 格式。2.14 的中间结果在 18.14 格式的 32 位累加器中累加。由于有 17 个守护位，除非 max(srcALen, srcBLen) 大于 131072，否则不会溢出。18.14 格式的结果丢弃低 7 位，截断为 18.7 格式，然后再饱和成 1.7 格式。

6.6 有限冲激响应（FIR）滤波器

6.6.1 函数

- *csi_fir_f32* : 浮点 FIR 滤波器处理函数
- *csi_fir_q31* : Q31 FIR 滤波器处理函数
- *csi_fir_q15* : Q15 FIR 滤波器处理函数
- *csi_fir_q7* : Q7 FIR 滤波器处理函数
- *csi_fir_fast_q31* : Q31 FIR 滤波器处理函数
- *csi_fir_fast_q15* : Q15 FIR 滤波器处理函数
- *csi_fir_init_f32* : 浮点 FIR 滤波器初始化函数
- *csi_fir_init_q31* : Q31 FIR 滤波器初始化函数
- *csi_fir_init_q15* : Q15 FIR 滤波器初始化函数
- *csi_fir_init_q7* : Q7 FIR 滤波器初始化函数

6.6.2 简要说明

这些函数实现了 Q7，Q15，Q31 和浮点数据类型的的有限冲激响应（FIR）滤波器，还有 Q15 和 Q31 的快速版本。还是以块为单位处理输入输出数据，每次调用滤波器函数处理 blockSize 个样本. pSrc 和 pDst 指向输入和输出数组，数组有 blockSize 个值。

算法:

FIR 滤波器的算法建立在一系列的乘加（MAC）操作之上。每个滤波器系数 b[n] 和一个状态变量相乘，状态变量与之前的输入样本 x[n] 相同。

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

pCoeffs 指向系数数组，数组的大小是 numTaps. 系数按以下顺序保存：

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

pState 指向状态数组，数组的大小是 numTaps + blockSize - 1. 状态数组的样本保存顺序如下：

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

注意：状态缓存的长度超过了系数数组的长度 blockSize-1. 扩展之后的状态缓存长度，可以避免循环寻址，并且显著提升速度。循环寻址是用在传统 FIR 滤波器的一种寻址方式。状态变量在每块数据操作之后更新，系数不更新。

结构体实例

滤波器的系数和状态变量都保存在数据结构的实例中。每个滤波器都必须有一个单独的结构体实例。系数数组可能可以在几个实例之间共享，但是状态变量数组不能共享。为支持的 4 种数据类型分别提供了不同的结构体实例声明。

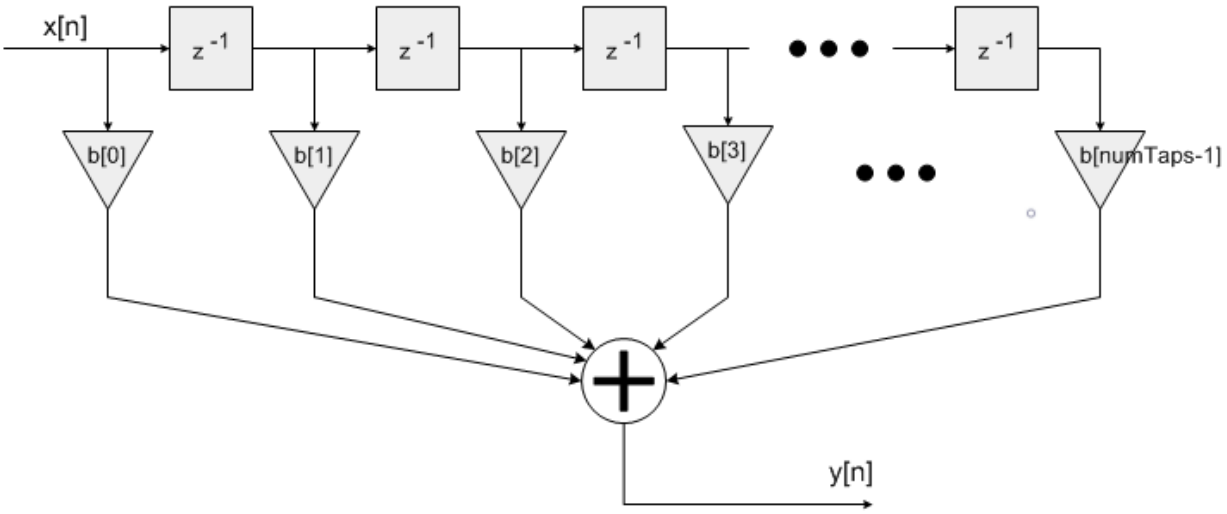


图 6.5: 有限冲激响应滤波器

初始化函数

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 结构体实例: 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 结构体实例 cannot be placed into a const data section. To place an 结构体实例 into a const data section, the 结构体实例 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 结构体实例 s 为每种支持的数据类型都提供了一个相应的初始化函数。初始化函数处理以下操作：

- 设置内部结构体字段的值
- 清零状态缓存中的值如果手动初始化，而不调用初始化函数，需要指定结构体实例的以下字段：numTaps, pCoeffs, pState. pState 中的所有值置 0.

是否使用初始化函数是可选的。但是，使用了初始化函数，则不能将结构体实例放在常量数据段。要将结构体实例放在常量数据段，则必须手动初始化结构体实例。在静态初始化之前，要确保状态缓存中的值已经清零。下面的代码，为 4 种不同的滤波器，静态的初始化了结构体实例。

```
*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};
```

其中 numTaps 是滤波器中的系数数量; pState 是状态缓存的地址; pCoeffs 是系数缓存的地址

定点行为

使用定点 FIR 滤波器函数需要注意。特别是要考虑，在每个函数内使用的累加器的溢出和饱和行为。具体参考每个函数各自的文档和使用说明。

6.6.3 函数说明

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)
```

参数:

`*S`: 指向 FIR 滤波器结构体实例

`*pSrc`: 指向输入数据

`*pDst`: 指向输出数据

`blockSize`: 输入数据的数量

返回值:

无

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)
```

参数:

`*S`: 指向 FIR 滤波器结构体实例

`*pSrc`: 指向输入数据

`*pDst`: 指向输出数据

`blockSize`: 输入数据的数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入数据表示为 1.31 格式。中间乘法生成 2.62 格式的结果，结果在 2.62 格式的 64 位累加器累加。因为只有一个保护位，所以需要缩小输入信号来防止溢出。需要缩小 log2(numTaps) 才可以保证没有溢出。最后，2.62 格式右移 31 位，然后饱和生成 1.15 格式的结果。

函数*csi_fir_fast_q31()* 是这个函数的一个快速版本，但是丢失了更多的精度

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)
```

参数:

*S: 指向 FIR 滤波器结构体实例

*pSrc: 指向输入数据

*pDst: 指向输出数据

blockSize: 输入数据的数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入都是 1.15 格式的，乘法生成 2.30 结果。2.30 格式的中间结果在 34.30 格式的 64 位累加器累加。由于有 33 个保护位，所以不会有溢出风险。34.30 的结果丢弃低 15 位截断为 34.15 格式，然后饱和成为 1.15 格式的结果。

函数*csi_fir_fast_q15()* 是这个函数的一个快速版本，但是丢失了更多的精度

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)
```

参数:

*S: 指向 FIR 滤波器结构体实例

*pSrc: 指向输入数据

*pDst: 指向输出数据

blockSize: 输入数据的数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 32 位累加器。输入都是 1.7 格式，相乘的结果是 2.14 格式。2.14 格式的结果在 18.14 格式的 32 位累加器中累加。中间结果不会有溢出风险，而且可以保留所有的精度。18.14 的结果丢弃低 7 位，截断为 18.7 格式，最后饱和为 1.7 格式。

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)
```

参数:

*S: 指向 FIR 滤波器结构体实例

*pSrc: 指向输入数据

*pDst: 指向输出数据

blockSize: 输入数据的数量

返回值:

无

缩放和溢出行为:

这个快速版本使用一个 2.30 格式的 32 位累加器。累加器维持了中间乘法的所有精度，但是只有一个保护位。因此为了防止溢出，输入信号必须缩小 log2(numTaps) 位. 最后 2.30 格式累加器截断为 2.15 格式，并且饱和为 1.15 格式.

函数*csi_fir_q15()* 是这个函数的一个慢速版本，使用了一个 64 位累加器，防止溢出。慢速和快速版本使用了相同的结构体实例。可以使用函数*csi_fir_init_q15()* 初始化滤波器结构体。

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)
```

参数:

*S: 指向 FIR 滤波器结构体实例

*pSrc: 指向输入数据

*pDst: 指向输出数据

blockSize: 输入数据的数量

返回值:

无

缩放和溢出行为:

函数为了优化速度，舍弃了一些定点精度和溢出保护。每个 1.31 和 1.31 相乘的结果截断为 2.30 格式. 这些中间结果在一个 2.30 格式累加器相加。最后，累加器饱和转换为 1.31 的结果。快速版本和标准版本有一样的溢出行为，因为丢弃了每次相乘结果的低 32 位，所以相对提供了更少的精度。为了防止溢出，输入信号必须缩小 log2(numTaps) 个位.

函数*csi_fir_q31()* 是这个函数的一个慢速版本，使用了一个 64 位累加器，提供了更高的精度。慢速和快速版本使用了相同的结构体实例。可以使用函数*csi_fir_init_q31()* 初始化滤波器的结构体.

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)
```

参数:

*S: 指向 FIR 滤波器结构体实例

numTaps: 滤波器内系数数量

*pCoeffs: 指向滤波器系数缓存

*pState: 指向状态缓存

blockSize: 输入数据的数量

返回值:

无

简要说明:

pCoeffs 指向滤波器系数数组，保存的顺序如下:

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

pState 指向状态变量的数组。pState 是长度为 numTaps+blockSize-1 的样本, 其中 blockSize 是传递给*csi_fir_f32()* 的输入样本数量.

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)
```

参数:

*S: 指向 FIR 滤波器结构体实例

numTaps: 滤波器内系数数量

*pCoeffs: 指向滤波器系数缓存

*pState: 指向状态缓存

blockSize: 输入数据的数量

返回值:

无

简要说明:

pCoeffs 指向滤波器系数数组，系数保存的顺序如下:

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

注意：numTaps 必须是偶数，并且大于等于 4。实现奇数长度的滤波器，则将 numTaps 加 1，然后将最后的系数设成 0。比如，要实现一个滤波器的 numTaps=3 ，系数是：

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

改成 numTaps=4 ，使用系数：

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

类似的，实现只有两个点的滤波器

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

改成 numTaps=4 ，使用系数：

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

pState 指向状态变量的数组。pState 的长度是 numTaps+blockSize，其中 blockSize 作为输入样本的数量传给*csi_fir_q31()* 。

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)
```

参数:

`*S`：指向 FIR 滤波器结构体实例

`numTaps`：滤波器内系数数量

`*pCoeffs`：指向滤波器系数缓存

`*pState`：指向状态缓存

`blockSize`：输入数据的数量

返回值:

无

简要说明:

pCoeffs 指向滤波器系数数组，系数保存的顺序如下：

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

pState 指向状态变量的数组。pState 的长度是 numTaps+blockSize-1 ，其中 blockSize 作为输入样本的数量传入*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)
```

参数:

`*S`: 指向 FIR 滤波器结构体实例

`numTaps`: 滤波器内系数数量

`*pCoeffs`: 指向滤波器系数缓存

`*pState`: 指向状态缓存

`blockSize`: 输入数据的数量

返回值:

无

简要说明:

pCoeffs 指向滤波器系数数组，系数保存的顺序如下:

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

pState 指向状态变量的数组。pState 的长度是 numTaps+blockSize-1，其中 blockSize 作为输入样本的数量传入*csi_fir_q7()* 。

6.7 有限冲激响应（FIR）抽取器

6.7.1 函数

- *csi_fir_decimate_f32* : 浮点 FIR 抽取处理函数
- *csi_fir_decimate_q31* : Q31 FIR 抽取处理函数
- *csi_fir_decimate_q15* : Q15 FIR 抽取处理函数
- *csi_fir_decimate_fast_q31* : Q31 FIR 抽取处理函数
- *csi_fir_decimate_fast_q15* : Q15 FIR 抽取处理函数
- *csi_fir_decimate_init_f32* : 浮点 FIR 抽取初始化函数
- *csi_fir_decimate_init_q31* : Q31 FIR 抽取初始化函数
- *csi_fir_decimate_init_q15* : Q15 FIR 抽取初始化函数

6.7.2 简要说明

这些函数将 FIR 滤波器和抽取器组合在一起。它们用于多速率系统中，用于降低信号的采样率而不引入混叠失真。从概念上讲，这些函数等同于下面的框图：

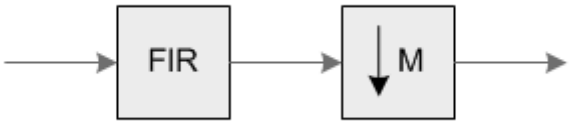


图 6.6: Components included in the FIR Decimator 函数

当用因子 M 抽取时, 信号应该通过 1/M 截止频率的低通滤波器预滤波，防止混叠失真。函数的使用者负责提供滤波器的系数。

CSI DSP 库中的 FIR 抽取器组合了 FIR 滤波器和抽取器。不会计算每个 M 的 FIR 滤波器输出，丢弃 M-1 , 只计算抽取器的输出样本。函数以块为单位操作输入和输出数据。pSrc 指向输入数组，数组大小是 blockSize ，pDst 指向输出数组，数组大小是 blockSize/M . 为了取得整数个输出样本 blockSize ， 输入必须是抽取因子 M 的整数倍数。

为 Q15，Q31 和浮点分别提供了不同的函数。

算法:

FIR 部分的算法使用的是标准形式过滤器:

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

其中, b[n] 是滤波器的系数。

pCoeffs 指向系数数组，数组大小是 numTaps. 系数按如下顺序排列保存：

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

pState 指向状态数组，数组大小是 numTaps + blockSize - 1. 样本在状态缓存中保存的顺序是:

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

状态变量会在每块数据处理后更新，系数不会更新。

结构体实例

滤波器的系数和状态变量都保存在数据结构的实例中。每个滤波器都必须要有一个结构体实例。系数数组可能可以在几个实例之间共享，但是状态变量数组必须单独分配。为浮点，Q31 和 Q15 数据类型分别提供了不同的结构体实例声明。

初始化函数

每种数据类型都有一个相应的初始化函数。初始化函数处理以下操作:

- 设置内部结构体字段的值
- 清零状态缓存
- 确保输入的大小是抽取因子的整数倍如果手动初始化，而不调用初始化函数，需要指定结构体实例的以下字段：numTaps, pCoeffs, M (抽取因子), pState. pState 中的所有值置 0.

是否使用初始化函数是可选的。但是，使用了初始化函数，则不能将结构体实例放在常量数据段。要将结构体实例放在常量数据段，则必须手动初始化结构体实例。下面的代码，为 3 种不同的滤波器，静态的初始化了结构体实例。

```
*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};
```

其中 M 是抽取因子; numTaps 是滤波器中的系数的数量；pCoeffs 是系数缓存的地址；pState 是状态缓存的地址。在静态初始化之前，要确保状态缓存中的值已经清零。

定点行为

使用定点 FIR 抽取器滤波函数需要注意。特别是要考虑，在每个函数内使用的累加器的溢出和饱和行为。具体参考每个函数各自的文档和使用说明。

6.7.3 函数说明

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)
```

参数:

*S：指向 FIR 抽取结构体实例

*pSrc：指向输入数据

*pDst：指向输出数据

blockSize：输入数据的数量

返回值:

无

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)
```

参数:

*S：指向 FIR 抽取结构体实例

*pSrc：指向输入数据

*pDst：指向输出数据

blockSize：输入数据的数量

返回值:

无

缩放和溢出行为:

快速版本实现使用了 2.30 格式的 32 位累加器。累加器维持了中间乘法的所有精度，但是只有一个保护位。因此如果累加器的结果溢出，会扭曲最后结果。为了防止溢出，输入信号需要缩小 log2(numTaps) 位 (log2 是 2 为底的对数) 最后，2.30 格式的丢弃低 15 位截断为 2.15，然后饱和生成 1.15 格式的结果。

函数*csi_fir_decimate_fast_q15()* 是这个函数的一个快速版本，但是丢失了更多的精度。

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)
```

参数:

*S：指向 FIR 抽取结构体实例

*pSrc：指向输入数据

*pDst：指向输出数据

blockSize：输入数据的数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入数据表示为 1.31 格式。中间乘法生成 2.62 格式的结果，结果在 2.62 格式的 64 位累加器累加。因为只有一个保护位，所以需要缩小输入信号来防止溢出。因为最多会有 numTaps 个加法进位，所以需要缩小 log2(numTaps) 才可以保证没有溢出。最后，2.62 格式右移 31 位，然后饱和生成 1.31 格式的结果。

函数*csi_fir_decimate_fast_q31()* 是这个函数的一个快速版本，但是丢失了更多的精度。

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)
```

参数:

`*S`: 指向 FIR 抽取结构体实例

`*pSrc`: 指向输入数据

`*pDst`: 指向输出数据

`blockSize`: 输入数据的数量

返回值:

无

限制:

如果芯片不支持分对齐访问，则输入，输出，临时 buffer，都应该是 32 位对齐。

缩放和溢出行为:

快速版本实现使用了 2.30 格式的 32 位累加器。累加器维持了中间乘法的所有精度，但是只有一个保护位。因此如果累加器的结果溢出，会扭曲最后结果。为了防止溢出，输入信号需要缩小 log2(numTaps) 位 (log2 是 2 为底的对数) 最后，2.30 格式的丢弃低 15 位截断为 2.15，然后饱和生成 1.15 格式的结果。

函数*csi_fir_decimate_q15()* 是这个函数的一个慢速版本，使用了 64 位累加器防止溢出，保留了更多的精度。慢速和快速版本使用相同的结构体实例。使用函数*csi_fir_decimate_init_q15()* 可以初始化滤波器结构体。

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)
```

参数:

*S：指向 FIR 抽取结构体实例

*pSrc：指向输入数据

*pDst：指向输出数据

blockSize：输入数据的数量

返回值:

无

缩放和溢出行为:

这个函数为了优化速度，舍弃了一些定点精度和溢出保护。每个 1.31 和 1.31 相乘的结果是 2.30 格式. 这些中间结果在一个 2.30 累加器相加。最后累加器饱和转换为 1.31 的结果。快速版本跟标准版本的有相同的溢出行为，由于丢弃了低 32 位相乘结果，维持了更少的精度。为了防止溢出，输入信号还需要缩小 log2(numTaps) 个位 (其中 log2 是 2 为底的对数).

函数*csi_fir_decimate_q31()* 是这个函数的一个慢速版本，使用了一个 64 位累加器，提供了更多的精度。慢速和快速版本使用相同的结构体实例。使用函数*csi_fir_decimate_init_q31()* 可以初始化滤波器结构体。

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)
```

参数:

*S：指向 FIR 抽取结构体实例

numTaps：滤波器系数的数量

M：抽取因子

*pCoeffs：指向滤波器系数

*pState：指向状态缓存

blockSize：输入样本的数量

返回值:

如果函数初始化成功，则返回 CSKY_MATH_SUCCESS ，如果 blockSize 不是 M 的整数倍，则返回 CSKY_MATH_LENGTH_ERROR.

简要说明:

pCoeffs 指向的滤波器系数数组，保存的顺序如下：

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

pState 指向状态变量数组， pState 的长度是 numTaps+blockSize-1 个字，其中 blockSize 是传递给*csi_fir_decimate_f32()* 的输入样本数量. M 是抽取因子.

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)
```

参数:

*S: 指向 FIR 抽取结构体实例

numTaps: 滤波器系数的数量

M: 抽取因子

*pCoeffs: 指向滤波器系数

*pState: 指向状态缓存

blockSize: 输入样本的数量

返回值:

如果函数初始化成功，则返回 CSKY_MATH_SUCCESS ，如果 blockSize 不是 M 的整数倍，则返回 CSKY_MATH_LENGTH_ERROR.

简要说明:

pCoeffs 指向的滤波器系数数组，保存的顺序如下:

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

pState 指向状态变量数组， pState 的长度是 numTaps+blockSize-1 个字，其中 blockSize 是传递给*csi_fir_decimate_q15()* 的输入样本数量. M 是抽取因子.

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)
```

参数:

*S: 指向 FIR 抽取结构体实例

numTaps: 滤波器系数的数量

M: 抽取因子

*pCoeffs: 指向滤波器系数

*pState: 指向状态缓存

blockSize: 输入样本的数量

返回值:

如果函数初始化成功，则返回 CSKY_MATH_SUCCESS ，如果 blockSize 不是 M 的整数倍，则返回 CSKY_MATH_LENGTH_ERROR.

简要说明:

pCoeffs 指向的滤波器系数数组，保存的顺序如下：

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

pState 指向状态变量数组， pState 的长度是 numTaps+blockSize-1 个字，其中 blockSize 是传递给*csi_fir_decimate_q31()* 的输入样本数量. M 是抽取因子.

6.8　有限冲激响应 (FIR) 格型滤波器

6.8.1　函数

- *csi_fir_lattice_f32* : 浮点 FIR 格型滤波器处理函数
- *csi_fir_lattice_q31* : Q31 FIR 格型滤波器处理函数
- *csi_fir_lattice_q15* : Q15 FIR 格型滤波器处理函数
- *csi_fir_lattice_init_f32* : 浮点 FIR 格型滤波器初始化函数
- *csi_fir_lattice_init_q31* : Q31 FIR 格型滤波器初始化函数
- *csi_fir_lattice_init_q15* : Q15 FIR 格型滤波器初始化函数

6.8.2　简要说明

这些函数实现了 Q15，Q31 和浮点的有限冲激响应 (FIR) 格型滤波器. 格型滤波器使用在各种自适应滤波器应用。滤波器结构是前馈的，并且净脉冲响应是有限长度。函数以块为单位操作输入输出数据，每次调用滤波器函数处理 blockSize 个样本. pSrc 和 pDst 指向输入输出数组，数组包括 blockSize 个值。

算法:

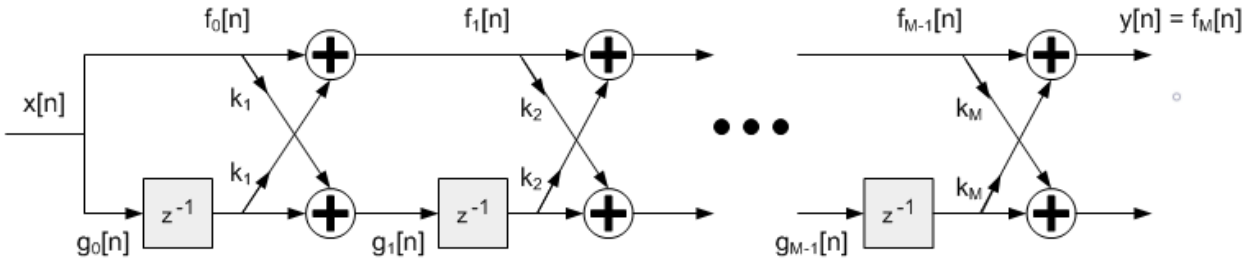


图 6.7: 有限冲激响应格型滤波器

实现了以下的差分方程:

```
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 指向反射系数数组，数组大小是 numStages. 反射系数保存的顺序如下:

```
{k1, k2, ..., kM}
```

其中 M 是阶段的序号。

pState 指向状态数组，数组的大小是 numStages. 状态变量 (g 的值) 维持了之前的输入，按以下的顺序保存:

```
{g0[n], g1[n], g2[n] ...gM-1[n]}
```

状态变量在每个块数据处理后更新，系数不会被更新。

结构体实例

滤波器的系数和状态变量都保存在数据结构的实例中。每个滤波器都必须有一个单独的结构体实例。系数数组可能可以在几个实例之间共享，但是状态变量数组不能共享。为支持的 3 种数据类型分别提供了不同的结构体实例声明。

初始化函数

为每种支持的数据类型都提供了一个相应的初始化函数。初始化函数处理以下操作：

- 设置内部结构体字段的值
- 清零状态缓存中的值如果手动初始化，而不调用初始化函数，需要指定结构体实例的以下字段：numStages, pCoeffs, pState. pState 中的所有值置 0.

是否使用初始化函数是可选的。但是，使用了初始化函数，则不能将结构体实例放在常量数据段。要将结构体实例放在常量数据段，则必须手动初始化结构体实例。在静态初始化之前，要确保状态缓存中的值已经清零。下面的代码，为 3 种不同的滤波器，静态的初始化了结构体实例。

```
*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};
```

其中是 numStages 滤波器阶段的数量; pState 是状态缓存的地址; pCoeffs 是系数缓存的地址。

定点行为

使用定点 FIR 格型滤波器函数需要注意。特别是要考虑，在每个函数内使用的累加器的溢出和饱和行为。具体参考每个函数各自的文档和使用说明。

6.8.3 函数说明

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)
```

参数:

`*S`: 指向 FIR 格型结构体实例

`*pSrc`: 指向输入数据

`*pDst`: 指向输出数据

`blockSize`: 输入数据的数量

返回值:

无

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)
```

参数:

*S: 指向 FIR 格型结构体实例
*pSrc: 指向输入数据
*pDst: 指向输出数据
blockSize: 输入数据的数量

返回值:

无

缩放和溢出行为:

为了防止溢出，输入信号必须缩小 2*log2(numStages) 个位。

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)
```

参数:

*S: 指向 FIR 格型结构体实例
*pSrc: 指向输入数据
*pDst: 指向输出数据
blockSize: 输入数据的数量

返回值:

无

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)
```

参数:

*S: 指向 FIR 格型结构体实例
numStages: 滤波器阶段数量
*pCoeffs: 指向系数缓存
*pState: 指向状态缓存

返回值:

无

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)
```

参数:

*S：指向 FIR 格型结构体实例

numStages：滤波器阶段数量

*pCoeffs：指向系数缓存

*pState：指向状态缓存

返回值:

无

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)
```

参数:

*S：指向 FIR 格型结构体实例

numStages：滤波器阶段数量

*pCoeffs：指向系数缓存

*pState：指向状态缓存

返回值:

无

6.9 有限冲激响应 (FIR) 稀疏滤波器

6.9.1 函数

- *csi_fir_sparse_f32* : 浮点稀疏 FIR 滤波器处理函数
- *csi_fir_sparse_q31* : Q31 稀疏 FIR 滤波器处理函数
- *csi_fir_sparse_q15* : Q15 稀疏 FIR 滤波器处理函数
- *csi_fir_sparse_q7* : Q7 稀疏 FIR 滤波器处理函数
- *csi_fir_sparse_init_f32* : 浮点稀疏 FIR 滤波器初始化函数
- *csi_fir_sparse_init_q31* : Q31 稀疏 FIR 滤波器初始化函数
- *csi_fir_sparse_init_q15* : Q15 稀疏 FIR 滤波器初始化函数
- *csi_fir_sparse_init_q7* : Q7 稀疏 FIR 滤波器初始化函数

6.9.2 简要说明

这些函数实现了稀疏 FIR 滤波器。

稀疏 FIR 滤波器跟标准 FIR 滤波器相似，只不过它的大多数系数值是 0。

稀疏滤波器常用在通信和音频应用中模拟反射。

为 Q7，Q15，Q31 和浮点数据类型分别提供了不同的函数。

函数以块为单位处理输入输出数据，并且滤波器每次调用处理 blockSize 个样本. pSrc 和 pDst 指向输入输出数组，数组分别包含 blockSize 个值。

算法:

稀疏滤波器结构体实例在系数数组 b 之外，还有一个索引 pTapDelay 来指定非零系数的位置。算法相比标准 FIR 效率更高的原因是，在算法实现里面利用了非零系数索引，省略了大多数与 0 相乘的步骤。

```
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 指向系数数组，数组的大小是 numTaps; pTapDelay 指向非零索引数组，数组的大小也是 numTaps; pState 指向状态数组，数组的大小 maxDelay + blockSize, 其中 maxDelay 是 pTapDelay 数组中最大的可用索引值。有些处理函数还要求提供一些临时的缓存。

结构体实例

滤波器的系数和状态变量都保存在数据结构的实例中。每个滤波器都必须有一个单独的结构体实例。系数数组可能可以在几个实例之间共享，但是状态变量数组不能共享。为支持的 4 种数据类型分别提供了不同的结构体实例声明。

初始化函数

为每种支持的数据类型都提供了一个相应的初始化函数。初始化函数处理以下操作：

- 设置内部结构体字段的值

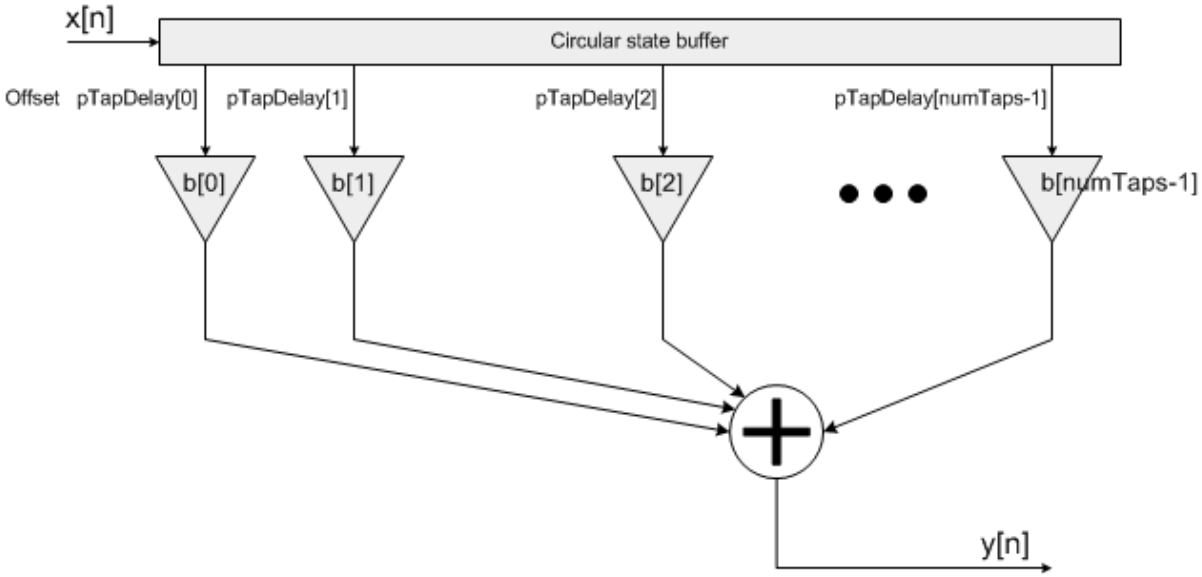


图 6.8: 稀疏 FIR 滤波器. b[n] 表示滤波器系数

- 清零状态缓存中的值如果手动初始化，而不调用初始化函数，需要指定结构体实例的以下字段：numTaps, pCoeffs, pTapDelay, maxDelay, stateIndex, pState. pState 中的所有值置 0.

是否使用初始化函数是可选的。但是，使用了初始化函数，则不能将结构体实例放在常量数据段。要将结构体实例放在常量数据段，则必须手动初始化结构体实例。在静态初始化之前，要确保状态缓存中的值已经清零。下面的代码，为 4 种不同的滤波器，静态的初始化了结构体实例。

```
*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};
```

定点行为

使用定点稀疏 FIR 滤波器函数需要注意。特别是要考虑，在每个函数内使用的累加器的溢出和饱和行为。具体参考每个函数各自的文档和使用说明。

6.9.3 函数说明

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)
```

参数:

*S: 指向稀疏 FIR 结构体实例

*pSrc: 指向输入数据

*pDst: 指向输出数据

*pScratchIn: 指向临时缓存，大小是 blockSize

blockSize: 指向处理的输入样本数量

返回值:

无

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)
```

参数:

*S: 指向稀疏 FIR 结构体实例

*pSrc: 指向输入数据

*pDst: 指向输出数据

*pScratchIn: 指向临时缓存，大小是 blockSize

blockSize: 指向处理的输入样本数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 32 位累加器。1.31 和 1.31 相乘的结果截断为 2.30 格式. 中间乘法的过程丢失了部分精度，并且只有一个保护位如果累加器溢出，不会做饱和处理，而是往符号位方向溢出。为了防止溢出，输入信号或者系数必须缩小 log2(numTaps) 位.

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)
```

参数:

*S：指向稀疏 FIR 结构体实例

*pSrc：指向输入数据

*pDst：指向输出数据

*pScratchIn：指向临时缓存，大小是 blockSize

*pScratchOut：指向临时缓存，大小是 blockSize

blockSize：指向处理的输入样本数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 32 位累加器。1.15 和 1.15 相乘生成 2.30 的结果，结果在 2.30 格式的累加器累加。因此相乘结果的所有精度都可以保留，但是累加器只有一个保护位。如果累加器溢出，不会做饱和处理，而是往符号位方向溢出。处理完所有的乘累加后，2.30 累加器截断成 2.15 格式，然后饱和成 1.15 格式。为了防止溢出，输入信号或者系数必须缩小 log2(numTaps) 位.

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)
```

参数:

*S：指向稀疏 FIR 结构体实例

*pSrc：指向输入数据

*pDst：指向输出数据

*pScratchIn：指向临时缓存，大小是 blockSize

*pScratchOut：指向临时缓存，大小是 blockSize

blockSize：指向处理的输入样本数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 32 位累加器。系数和状态变量都用 1.7 格式表示，相乘生成 2.14 结果。2.14 的中间结果在 18.14 格式的 32 位累加器累加。这些操作可以保留所有的精度，并且不会有溢出风险累加器之后丢弃低 7 位截断为 18.7 格式。最后，结果转换为 1.7 格式。

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)
```

参数:

*S: 指向稀疏 FIR 结构体实例
numTaps: 滤波器中非零系数数量
*pCoeffs: 指向滤波器系数数组
*pState: 指向状态数组
*pTapDelay: 指向偏移数组
maxDelay: 支持的最大偏移
blockSize: 处理的样本数量

返回值:

无

简要说明:

pCoeffs 保存了滤波器系数，长度为 numTaps. pState 保存了滤波器的状态变量，长度为 maxDelay + blockSize, 其中 maxDelay 是偏移支持的最大值。blockSize 是处理样本的数量，传入给*csi_fir_sparse_f32()* .

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)
```

参数:

*S: 指向稀疏 FIR 结构体实例
numTaps: 滤波器中非零系数数量
*pCoeffs: 指向滤波器系数数组
*pState: 指向状态数组
*pTapDelay: 指向偏移数组
maxDelay: 支持的最大偏移
blockSize: 处理的样本数量

返回值:

无

简要说明:

pCoeffs 保存了滤波器系数，长度为 numTaps. pState 保存了滤波器的状态变量，长度为 maxDelay + blockSize, 其中 maxDelay 是偏移支持的最大值。blockSize 是处理样本的数量，传入给*csi_fir_sparse_q31()* 。

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)
```

参数:

*S: 指向稀疏 FIR 结构体实例
numTaps: 滤波器中非零系数数量
*pCoeffs: 指向滤波器系数数组
*pState: 指向状态数组
*pTapDelay: 指向偏移数组
maxDelay: 支持的最大偏移
blockSize: 处理的样本数量

返回值:

无

简要说明:

pCoeffs 保存了滤波器系数，长度为 numTaps. pState 保存了滤波器的状态变量，长度为 maxDelay + blockSize, 其中 maxDelay 是偏移支持的最大值。blockSize 是处理样本的数量，传入给*csi_fir_sparse_q15()* 。

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)
```

参数:

*S: 指向稀疏 FIR 结构体实例
numTaps: 滤波器中非零系数数量
*pCoeffs: 指向滤波器系数数组
*pState: 指向状态数组
*pTapDelay: 指向偏移数组
maxDelay: 支持的最大偏移
blockSize: 处理的样本数量

返回值:

无

简要说明:

pCoeffs 保存了滤波器系数，长度为 numTaps. pState 保存了滤波器的状态变量，长度为 maxDelay + blockSize, 其中 maxDelay 是偏移支持的最大值。blockSize 是处理样本的数量，传入给*csi_fir_sparse_q7()* 。

6.10　有限冲激响应 (FIR) 插值滤波器

6.10.1　函数

- *csi_fir_interpolate_f32* : 浮点 FIR 插值处理函数
- *csi_fir_interpolate_q31* : Q31 FIR 插值处理函数
- *csi_fir_interpolate_q15* : Q15 FIR 插值处理函数
- *csi_fir_interpolate_init_f32* : 浮点 FIR 插值初始化函数
- *csi_fir_interpolate_init_q31* : Q31 FIR 插值初始化函数
- *csi_fir_interpolate_init_q15* : Q15 FIR 插值初始化函数

6.10.2　简要说明

函数组合了一个过采样器 (零填充) 和一个 FIR 滤波器。

这些函数用于多速率系统中，在不引入高频图像的情况下增加信号的采样率。

从概念上讲，这些功能等同于下面的框图：

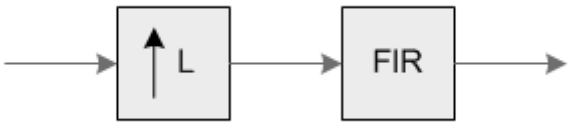


图 6.9: FIR 插值器中的组件

通过因子 L 过采样后, 信号需要用一个截止频率 1/L 的低通滤波器滤波，以便消除频谱的高频拷贝。函数的调用者负责提供滤波器的系数。

CSI DSP 库中的 FIR 插值函数提供了过采样器和 FIR 滤波器的一种组合方式。过采样器在样本中插入了 L-1 个 0. 滤波器设计成不会与这些插值出来的 0 相乘，而是直接跳过，这样可以提高效率，减少计算量。函数以块为单位操作输入输出数据。pSrc 指向一个大小为 blockSize 的输入数组，pDst 指向一个大小为 blockSize*L 的输出数组.

DSP 库为 Q15，Q31 和浮点数据类型分别提供了不同的函数。

算法:

函数使用一个多相滤波器结构:

```
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]
...
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]
```

这种方式比直接过采样，然后滤波的算法更效率。这种方式的计算量减少到标准 FIR 滤波器的 1/L.

pCoeffs 指向一个系数数组，数组大小为 numTaps. numTaps 必须是插值因子的 L 的倍数，并且会在初始化函数里检查。函数内部会将 FIR 滤波器的冲激响应除成更短的长度 phaseLength=numTaps/L. 系数的保存顺序如下：

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

pState 指向状态数组，数组的大小为 blockSize + phaseLength - 1. 状态缓存中的保存顺序如下:

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

状态变量在数据块处理的时候会更新，系数不会更新。

结构体实例

滤波器的系数和状态变量都保存在数据结构的实例中。每个滤波器都必须要有一个结构体实例。系数数组可能可以在几个实例之间共享，但是状态变量数组必须单独分配。为浮点，Q31 和 Q15 数据类型分别提供了不同的结构体实例声明。

初始化函数

每种数据类型都有一个相应的初始化函数。初始化函数处理以下操作：

- 设置内部结构体字段的值
- 清零状态缓存
- 确保滤波器的长度是插值因子的整数倍如果手动初始化，而不调用初始化函数，需要指定结构体实例的以下字段：L (插值因子), pCoeffs, phaseLength (numTaps / L), pState. pState 中的所有值置 0

是否使用初始化函数是可选的。但是，使用了初始化函数，则不能将结构体实例放在常量数据段。要将结构体实例放在常量数据段，则必须手动初始化结构体实例。下面的代码，为 3 种不同的滤波器，静态的初始化了结构体实例。

```
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};
```

其中 L 是插值因子; phaseLength=numTaps/L 是 FIR 滤波器内部使用的更短的长度，pCoeffs 是系数缓存的地址; pState 是状态缓存的地址. 在静态初始化之前，要确保状态缓存中的值已经清零。

定点行为

使用定点 FIR 插值器滤波函数需要注意。特别是要考虑，在每个函数内使用的累加器的溢出和饱和行为。具体参考每个函数各自的文档和使用说明。

6.10.3 函数说明

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)
```

参数:

*S: 指向 FIR 插值结构体实例

*pSrc: 指向输入数据

*pDst: 指向输出数据

blockSize: 输入数据的数量

返回值:

无

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)
```

参数:

*S: 指向 FIR 插值结构体实例

*pSrc: 指向输入数据

*pDst: 指向输出数据

blockSize: 输入数据的数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入数据表示为 1.31 格式。中间乘法生成 2.62 格式的结果，结果在 2.62 格式的 64 位累加器累加。因为只有一个保护位，所以需要缩小输入信号来防止溢出。因为最多会有 numTaps/L 个加法进位，所以需要缩小 1/(numTaps/L) 才可以保证没有溢出。最后，2.62 格式右移 31 位，然后饱和生成 1.15 格式的结果。

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)
```

参数:

*S：指向 FIR 插值结构体实例

*pSrc：指向输入数据

*pDst：指向输出数据

blockSize：输入数据的数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入数据表示为 1.15 格式。中间乘法生成 2.30 格式的结果，结果在 34.30 格式的 64 位累加器累加。因为有 33 位保护位，所以不会有溢出的风险。同时还可以保存所有的中间乘法结果的精度。最后，34.30 格式的丢弃低 15 位截断为 34.15，然后饱和生成 1.15 格式的结果。

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)
```

参数:

*S：指向 FIR 插值结构体实例

L：过采样因子

numTaps：滤波器中系数的数量

*pCoeffs：指向系数缓存

*pState：指向状态缓存

blockSize：处理的输入样本数量

返回值:

无

简要说明:

pCoeffs 指向滤波器系数数组，保存的顺序如下：

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

滤波器的长度 numTaps 必须是插值因子 L 的整数倍.

pState 指向状态变量数组。pState 的长度是 (numTaps/L)+blockSize-1 。其中 blockSize 是处理的输入样本数量，传给*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)
```

参数:

*S: 指向 FIR 插值结构体实例

L: 过采样因子

numTaps: 滤波器中系数的数量

*pCoeffs: 指向系数缓存

*pState: 指向状态缓存

blockSize: 处理的输入样本数量

返回值:

无

简要说明:

pCoeffs 指向滤波器系数数组，保存的顺序如下:

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

滤波器的长度 numTaps 必须是插值因子 L 的整数倍.

pState 指向状态变量数组。pState 的长度是 (numTaps/L)+blockSize-1 。其中 blockSize 是处理的输入样本数量，传给*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)
```

参数:

*S: 指向 FIR 插值结构体实例

L: 过采样因子

numTaps: 滤波器中系数的数量

*pCoeffs: 指向系数缓存

*pState: 指向状态缓存

blockSize: 处理的输入样本数量

返回值:

无

简要说明:

pCoeffs 指向滤波器系数数组，保存的顺序如下:

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

滤波器的长度 numTaps 必须是插值因子 L 的整数倍.

pState 指向状态变量数组。pState 的长度是 (numTaps/L)+blockSize-1 。其中 blockSize 是处理的输入样本数量，传给*csi_fir_interpolate_q15()* 。

6.11　无限冲激响应 (IIR) 格型滤波器

6.11.1　函数

- *csi_iir_lattice_f32* : 浮点 IIR 格型滤波器处理函数
- *csi_iir_lattice_q31* : Q31 IIR 格型滤波器处理函数
- *csi_iir_lattice_q15* : Q15 IIR 格型滤波器处理函数
- *csi_iir_lattice_init_f32* : 浮点 IIR 格型滤波器初始化函数
- *csi_iir_lattice_init_q31* : Q31 IIR 格型滤波器初始化函数
- *csi_iir_lattice_init_q15* : Q15 IIR 格型滤波器初始化函数

6.11.2　简要说明

这些函数实现了 Q15，Q31 和浮点的无限冲激响应 (IIR) 格型滤波器. 格型滤波器使用在各种自适应滤波器应用。滤波器结构具有前馈和反馈分量，并且净脉冲响应是无限长度。函数以块为单位操作输入输出数据，每次调用滤波器函数处理 blockSize 个样本. pSrc 和 pDst 指向输入输出数组，数组包括 blockSize 个值。

算法:

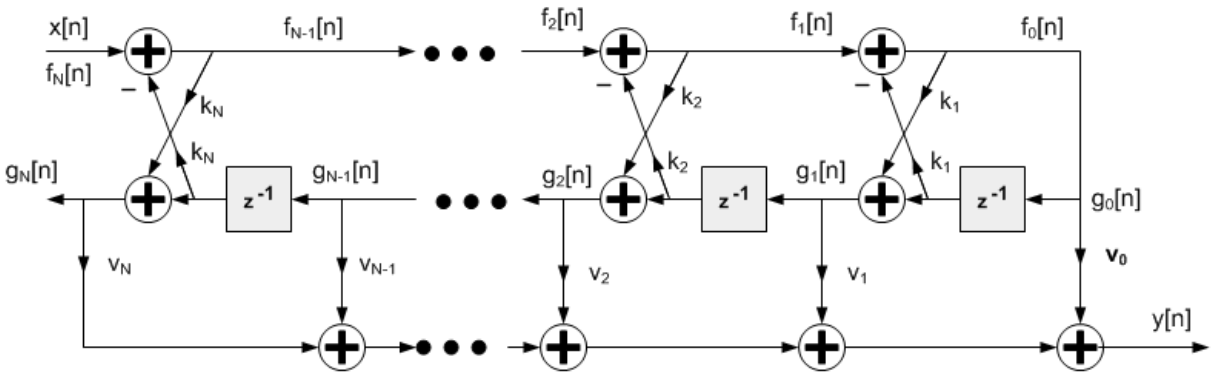


图 6.10: 无限冲激响应格型滤波器

```
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 指向反射系数数组，数组大小是 numStages. 反射系数保存的顺序如下:

```
{kN, kN-1, ....k1}
```

pvCoeffs 指向梯形系数数组，数组大小是 (numStages+1). 梯形系数数组保存的顺序如下:

```
{vN, vN-1, ...v0}
```

pState 指向状态数组，数组大小是 numStages + blockSize. 上面提到的状态变量（g 值）保存在 pState 数组. 状态变量在每个块数据处理后更新，系数不会被更新。

结构体实例

滤波器的系数和状态变量都保存在数据结构的实例中。每个滤波器都必须有一个单独的结构体实例。系数数组可能可以在几个实例之间共享，但是状态变量数组不能共享。为支持的 3 种数据类型分别提供了不同的结构体实例声明。

初始化函数

为每种支持的数据类型都提供了一个相应的初始化函数。初始化函数处理以下操作：

- 设置内部结构体字段的值
- 清零状态缓存中的值如果手动初始化，而不调用初始化函数，需要指定结构体实例的以下字段：numStages, pkCoeffs, pvCoeffs, pState. pState 中的所有值置 0.

是否使用初始化函数是可选的。但是，使用了初始化函数，则不能将结构体实例放在常量数据段。要将结构体实例放在常量数据段，则必须手动初始化结构体实例。在静态初始化之前，要确保状态缓存中的值已经清零。下面的代码，为 3 种不同的滤波器，静态的初始化了结构体实例。

```
*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};
```

其中 numStages 是滤波器阶段的数量; pState 指向状态缓存数组; pkCoeffs 指向反射系数数组; pvCoeffs 指向梯形系数数组.

定点行为

使用定点 IIR 格型滤波器函数需要注意。特别是要考虑，在每个函数内使用的累加器的溢出和饱和行为。具体参考每个函数各自的文档和使用说明。

6.11.3 函数说明

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)
```

参数:

`*S`: 指向 IIR 格型结构体实例

`*pSrc`: 指向输入数据

`*pDst`: 指向输出数据

`blockSize`: 输入数据的数量

返回值:

无

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)
```

参数:

*S: 指向 IIR 格型结构体实例

*pSrc: 指向输入数据

*pDst: 指向输出数据

blockSize: 输入数据的数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入数据表示为 1.31 格式。中间乘法生成 2.62 格式的结果，结果在 2.62 格式的 64 位累加器累加。因为只有一个保护位，所以需要缩小输入信号来防止溢出。为了防止溢出，必须缩小 2*log2(numStages) 个位。最后，2.62 格式右移 31 位，然后饱和生成 1.15 格式的结果。

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)
```

参数:

*S: 指向 IIR 格型结构体实例

*pSrc: 指向输入数据

*pDst: 指向输出数据

blockSize: 输入数据的数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。系数和状态变量都表示为 1.15 格式。中间乘法生成 2.30 格式的结果，结果在 34.30 格式的 64 位累加器累加。因为有 33 位保护位，所以不会有溢出的风险。同时还可以保存所有的中间乘法结果的精度。最后，34.30 格式的丢弃低 15 位截断为 34.15，然后饱和生成 1.15 格式的结果。

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)
```

参数:

*S：指向 IIR 格型结构体实例

numStages：滤波器阶段的数量

*pkCoeffs：指向反射系数缓存，数组的长度是 numStages

*pvCoeffs：指向梯形系数缓存，数组的长度是 numStages+1

*pState：指向状态缓存，数组的长度是 numStages+blockSize

blockSize：处理的样本数量

返回值:

无

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)
```

参数:

*S：指向 IIR 格型结构体实例

numStages：滤波器阶段的数量

*pkCoeffs：指向反射系数缓存，数组的长度是 numStages

*pvCoeffs：指向梯形系数缓存，数组的长度是 numStages+1

*pState：指向状态缓存，数组的长度是 numStages+blockSize

blockSize：处理的样本数量

返回值:

无

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)
```

参数:

*S：指向 IIR 格型结构体实例

numStages：滤波器阶段的数量

*pkCoeffs：指向反射系数缓存，数组的长度是 numStages

*pvCoeffs：指向梯形系数缓存，数组的长度是 numStages+1

*pState：指向状态缓存，数组的长度是 numStages+blockSize

blockSize：处理的样本数量

返回值:

无

6.12　最小均方 (LMS) 滤波器

6.12.1　函数

- *csi_lms_f32* : 浮点 LMS 滤波器处理函数
- *csi_lms_q31* : Q31 LMS 滤波器处理函数
- *csi_lms_q15* : Q15 LMS 滤波器处理函数
- *csi_lms_init_f32* : 浮点 LMS 滤波器初始化函数
- *csi_lms_init_q31* : Q31 LMS 滤波器初始化函数
- *csi_lms_init_q15* : Q15 LMS 滤波器初始化函数

6.12.2　简要说明

LMS 滤波器是一类自适应滤波器，使用的是一种梯度下降的算法，根据瞬时误差更新滤波器的系数，达到“学习”一种未知的转换方式目的。

自适应滤波器常应用在通信系统，均衡器，和噪声去除。

CSI DSP 库内的 LMS 滤波器函数支持 Q15, Q31 和浮点数据类型。

库内也有归一化 LMS 滤波器，归一化 LMS 滤波器系数自适应与输入信号的电平无关。

一个 LMS 滤波器包括以下两个部分:

- 第一部分是一个标准横向 FIR 滤波器。
- 第二部分是系数更新机制。

LMS 滤波器有两个输入信号。一个是接受的输入信号，另一个是参考的输入信号，输出两个信号，一个是 FIR 滤波器的输出信号，另一个是与参考输入相比的误差信号。滤波器根据输出和参考输入之间的差值更新系数，直到 FIR 滤波器的 FIR 滤波器的输出跟参考输入相符。误差通过滤波器的调解倾向于 0。

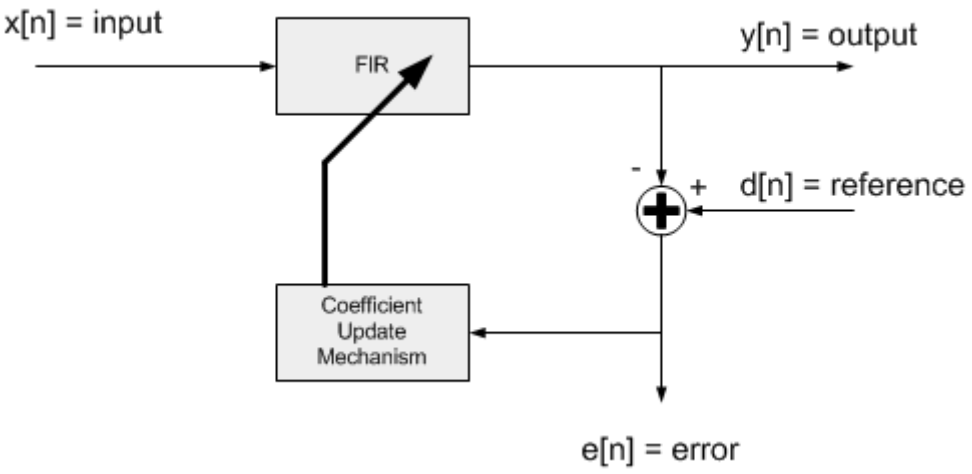


图 6.11: 最小均方滤波器结构

函数以块为单位处理数据，每次调用滤波器函数处理 blockSize 个样本。pSrc 指向输入信号, pRef 指向参考信号，pOut 指向输出信号和 pErr 指向误差信号。所有的数组都包括 blockSize 个值。

函数以块为单位操作。滤波器内部系数 b[n] 以样本为单位更新。

算法:

输出信号 y[n] 通过标准 FIR 滤波器计算:

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

误差信号等于参考信号 d[n] 和滤波器输出的差值:

```
e[n] = d[n] - y[n].
```

计算每个误差信号的每个样本后，滤波器系数 b[k] 以样本为单位更新:

```
b[k] = b[k] + e[n] * mu * x[n-k],  for k=0, 1, ..., numTaps-1
```

其中 mu 是步进大小，控制系数收敛速率。

接口中, pCoeffs 指向系数数组，大小是 numTaps. 系数保存的顺序如下:

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

pState 指向状态数组，数组大小是 numTaps + blockSize - 1. 样本在状态缓存中的保存顺序是:

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

注意：状态缓存的长度超过了系数数组 blockSize-1 个样本. 增长的状态缓存长度可以用来取代传统 FIR 滤波器使用的循环寻址，从而显著提高速度。状态变量在每块数据处理后更新。

结构体实例

滤波器的系数和状态变量都保存在数据结构的实例中。每个滤波器都必须有一个单独的结构体实例。系数数组可能可以在几个实例之间共享，但是状态变量数组不能共享。为支持的 3 种数据类型分别提供了不同的结构体实例声明。

初始化函数

为每种支持的数据类型都提供了一个相应的初始化函数。初始化函数处理以下操作:

- 设置内部结构体字段的值
- 清零状态缓存中的值如果手动初始化，而不调用初始化函数，需要指定结构体实例的以下字段：numTaps, pCoeffs, mu, postShift (f32 不需要), pState. pState 中的所有值置 0

是否使用初始化函数是可选的。但是，使用了初始化函数，则不能将结构体实例放在常量数据段。要将结构体实例放在常量数据段，则必须手动初始化结构体实例。在静态初始化之前，要确保状态缓存中的值已经清零。下面的代码，为 3 种不同的滤波器，静态的初始化了结构体实例。

```
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};
```

其中 numTaps 是滤波器系数的数量; pState 是状态缓存的地址; pCoeffs 是系数缓存的地址; mu 是步进大小; postShift 是系数的移位数.

定点行为:

使用 Q15 和 Q31 版本的 LMS 滤波器函数需要注意。下列问题必须考虑:

- 系数缩放
- 溢出和饱和

系数缩放:

滤波器系数表示为一个小数值，被限制在范围 [-1 +1) 之间。定点函数有一个附加的缩放参数 postShift。滤波器的输出累加器是一个可移位的寄存器，结果移动 postShift 位。基本上就是将滤波器系数缩放 2^postShift，允许将滤波器的系数扩展到超过范围 [+1 -1)。postShift 的值根据用户系统模型期望的增益设定。

溢出和饱和:

Q15 和 Q31 版本的溢出和饱和分别描述在各个函数各自的文档部分。

6.12.3 函数说明

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)
```

参数:

`*S`: 指向 LMS 滤波器结构体实例

`*pSrc`: 指向输入数据

`*pRef`: 指向参考数据

`*pOut`: 指向输出数据

`*pErr`: 指向误差数据

`blockSize`: 处理的样本的数量

返回值:

无

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)
```

参数:

*S：指向 LMS 滤波器结构体实例

*pSrc：指向输入数据

*pRef：指向参考数据

*pOut：指向输出数据

*pErr：指向误差数据

blockSize：处理的样本的数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。累加器是 2.62 格式，并且维持了中间乘法结果的所有精度，但是只有一个保护位。为了防止溢出，输入信号必须缩小 log2(numTaps) 位. 参考信号不应该缩小。在所有的乘累加处理后，2.62 格式累加器右移，然后饱和生成 1.31 最后的结果输出信号和错误信号是 1.31 格式.

这个滤波器中，滤波器系数会根据样本更新，并且系数更新是饱和的。

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)
```

参数:

*S：指向 LMS 滤波器结构体实例

*pSrc：指向输入数据

*pRef：指向参考数据

*pOut：指向输出数据

*pErr：指向误差数据

blockSize：处理的样本的数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。系数和状态变量都表示为 1.15 格式。中间乘法生成 2.30 格式的结果，结果在 34.30 格式的 64 位累加器累加。因为有 33 位保护位，所以不会有溢出的风险。同时还可以保存所有的中间乘法结果的精度。最后，34.30 格式的丢弃低 15 位截断为 34.15，然后饱和生成 1.15 格式的结果。

这个滤波器中，滤波器系数会根据样本更新，并且系数更新是饱和的。

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)
```

参数:

*S: 指向 LMS 滤波器结构体实例

numTaps: 滤波器系数的数量

*pCoeffs: 指向系数缓存

*pState: 指向状态缓存

mu: 控制滤波器系数更新的步进大小

blockSize: 处理的样本的数量

返回值:

无

简要说明:

pCoeffs 指向滤波器系数的数组，保存的顺序如下:

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

初始化滤波器系数作为自适应滤波器的起始点 pState 指向一个长度为 numTaps+blockSize-1 样本数组, 其中 blockSize 是处理的输入样本数量，传入函数*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)
```

参数:

*S: 指向 LMS 滤波器结构体实例

numTaps: 滤波器系数的数量

*pCoeffs: 指向系数缓存

*pState: 指向状态缓存

mu: 控制滤波器系数更新的步进大小

blockSize: 处理的样本的数量

postShift: 系数的移位数

返回值:

无

简要说明:

pCoeffs 指向滤波器系数的数组，保存的顺序如下:

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

初始化滤波器系数作为自适应滤波器的起始点 pState 指向一个长度为 numTaps+blockSize-1 样本数组, 其中 blockSize 是处理的输入样本数量，传入函数*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)
```

参数:

`*S`: 指向 LMS 滤波器结构体实例

`numTaps`: 滤波器系数的数量

`*pCoeffs`: 指向系数缓存

`*pState`: 指向状态缓存

`mu`: 控制滤波器系数更新的步进大小

`blockSize`: 处理的样本的数量

`postShift`: 系数的移位数

返回值:

无

简要说明:

pCoeffs 指向滤波器系数的数组，保存的顺序如下:

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

初始化滤波器系数作为自适应滤波器的起始点 pState 指向一个长度为 numTaps+blockSize-1 样本数组, 其中 blockSize 是处理的输入样本数量，传入函数*csi_lms_q15()* 。

6.13　归一化 LMS 滤波器

6.13.1　函数

- *csi_lms_norm_f32* : 浮点归一化 LMS 滤波器处理函数
- *csi_lms_norm_q31* : Q31 归一化 LMS 滤波器处理函数
- *csi_lms_norm_q15* : Q15 归一化 LMS 滤波器处理函数
- *csi_lms_norm_init_f32* : 浮点归一化 LMS 滤波器初始化函数
- *csi_lms_norm_init_q31* : Q31 归一化 LMS 滤波器初始化函数
- *csi_lms_norm_init_q15* : Q15 归一化 LMS 滤波器初始化函数

6.13.2　简要说明

这组函数实现了常用的自适应滤波器。归一化 LMS 在最小均方 (LMS) 自适应滤波器的基础上，附加了额外的归一化因子，提高了滤波器的自适应速率。CSI DSP 库内的归一化 LMS 滤波器函数支持 Q15, Q31 和浮点数据类型。

一个归一化最小均方 (NLMS) 滤波器包括以下两部分。

- 第一部分是一个标准横向 FIR 滤波器。
- 第二部分是系数更新机制。

归一化 LMS 滤波器有两个输入信号。一个是接受的输入信号，另一个是参考的输入信号，输出两个信号，一个是 FIR 滤波器的输出信号，另一个是与参考输入相比的误差信号。滤波器根据输出和参考输入之间的差值更新系数，直到 FIR 滤波器的 FIR 滤波器的输出跟参考输入相符。误差通过滤波器的调解倾向于 0。

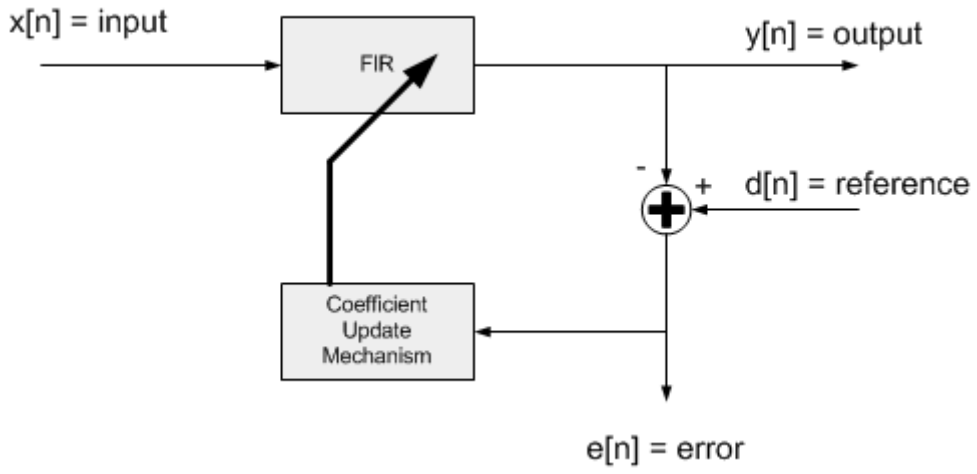


图 6.12: 归一化 LMS 滤波器中间结构

函数以块为单位处理数据，每次调用滤波器函数处理 blockSize 个样本。pSrc 指向输入信号, pRef 指向参考信号，pOut 指向输出信号和 pErr 指向误差信号。所有的数组都包括 blockSize 个值。

函数以块为单位操作。滤波器内部系数 b[n] 以样本为单位更新。

算法:

输出信号 y[n] 通过标准 FIR 滤波器计算:

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

误差信号等于参考信号 d[n] 和滤波器输出的差值:

```
e[n] = d[n] - y[n].
```

计算每个误差信号的每个样本后，滤波器状态变量的瞬时能量:

```
E = x[n]^2 + x[n-1]^2 + ... + x[n-numTaps+1]^2.
```

滤波器系数 b[k] 以样本为单位更新:

```
b[k] = b[k] + e[n] * (mu/E) * x[n-k],  for k=0, 1, ..., numTaps-1
```

其中 mu 是步进大小，控制系数收敛速率。

接口中, pCoeffs 指向系数数组，大小是 numTaps. 系数保存的顺序如下:

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

pState 指向状态数组，数组大小是 numTaps + blockSize - 1. 样本在状态缓存中的保存顺序是:

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

注意：状态缓存的长度超过了系数数组 blockSize-1 个样本. 增长的状态缓存长度可以取代传统 FIR 滤波器使用的循环寻址，显著提高速度。状态变量在每块数据处理后更新。

结构体实例

滤波器的系数和状态变量都保存在数据结构的实例中。每个滤波器都必须有一个单独的结构体实例。系数数组可能可以在几个实例之间共享，但是状态变量数组不能共享。为支持的 3 种数据类型分别提供了不同的结构体实例声明。

初始化函数

为每种支持的数据类型都提供了一个相应的初始化函数。初始化函数处理以下操作:

- 设置内部结构体字段的值
- 清零状态缓存中的值如果手动初始化，而不调用初始化函数，需要指定结构体实例的以下字段：numTaps, pCoeffs, mu, energy, x0, pState. pState 中的所有值置 0. 对 Q7, Q15, 和 Q31 类型，下列字段必须被初始化：recipTable, postShift

结构体实例不能被放入常量数据段，推荐使用初始化函数初始化它。

定点行为:

使用 Q15 和 Q31 版本的归一化 LMS 滤波器函数需要注意。下列问题必须考虑:

- 系数缩放

- 溢出和饱和

系数缩放:

滤波器系数表示为一个小数值，被限制在范围 [-1 +1) 之间。定点函数有一个附加的缩放参数 postShift。滤波器的输出累加器是一个可移位的寄存器，结果移动 postShift 位。基本上就是将滤波器系数缩放 2^postShift，允许将滤波器的系数扩展到超过范围 [+1 -1)。postShift 的值根据用户系统模型期望的增益设定

溢出和饱和:

Q15 和 Q31 版本的溢出和饱和分别描述在各个函数各自的文档部分。

6.13.3 函数说明

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)
```

参数:

*S: 指向 LMS 滤波器结构体实例

*pSrc: 指向输入数据

*pRef: 指向参考数据

*pOut: 指向输出数据

*pErr: 指向误差数据

blockSize: 处理的样本的数量

返回值:

无

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)
```

参数:

*S: 指向 LMS 滤波器结构体实例

*pSrc: 指向输入数据

*pRef: 指向参考数据

*pOut: 指向输出数据

*pErr: 指向误差数据

blockSize: 处理的样本的数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。累加器是 2.62 格式，并且维持了中间乘法结果的所有精度，但是只有一个保护位。为了防止溢出，输入信号必须缩小 log2(numTaps) 位. 参考信号不应该缩小。在所有的乘累加处理后，2.62 格式累加器右移，然后饱和生成 1.31 最后的结果输出信号和错误信号是 1.31 格式。

这个滤波器中，滤波器系数会根据样本更新，并且系数更新是饱和的。

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)
```

参数:

`*S`: 指向 LMS 滤波器结构体实例

`*pSrc`: 指向输入数据

`*pRef`: 指向参考数据

`*pOut`: 指向输出数据

`*pErr`: 指向误差数据

`blockSize`: 处理的样本的数量

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。系数和状态变量都表示为 1.15 格式。中间乘法生成 2.30 格式的结果，结果在 34.30 格式的 64 位累加器累加。因为有 33 位保护位，所以不会有溢出的风险。同时还可以保存所有的中间乘法结果的精度。最后，34.30 格式的丢弃低 15 位截断为 34.15，然后饱和生成 1.15 格式的结果。

这个滤波器中，滤波器系数会根据样本更新，并且系数更新是饱和的。

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)
```

参数:

`*S`: 指向 LMS 滤波器结构体实例

`numTaps`: 滤波器系数的数量

`*pCoeffs`: 指向系数缓存

*pState: 指向状态缓存

mu: 控制滤波器系数更新的步进大小

blockSize: 处理的样本的数量

返回值:

无

缩放和溢出行为:

pCoeffs 指向滤波器系数的数组，保存的顺序如下：

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

初始化滤波器系数作为自适应滤波器的起始点。pState 指向一个长度为 numTaps+blockSize-1 样本数组, 其中 blockSize 是处理的输入样本数量，传入函数*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)
```

参数:

*S: 指向 LMS 滤波器结构体实例

numTaps: 滤波器系数的数量

*pCoeffs: 指向系数缓存

*pState: 指向状态缓存

mu: 控制滤波器系数更新的步进大小

blockSize: 处理的样本的数量

postShift: 系数的移位数

返回值:

无

缩放和溢出行为:

pCoeffs 指向滤波器系数的数组，保存的顺序如下：

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

初始化滤波器系数作为自适应滤波器的起始点。pState 指向一个长度为 numTaps+blockSize-1 样本数组, 其中 blockSize 是处理的输入样本数量，传入函数*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)
```

参数:

*S: 指向 LMS 滤波器结构体实例

numTaps: 滤波器系数的数量

*pCoeffs: 指向系数缓存

*pState: 指向状态缓存

mu: 控制滤波器系数更新的步进大小

blockSize: 处理的样本的数量

postShift: 系数的移位数

返回值:

无

缩放和溢出行为:

pCoeffs 指向滤波器系数的数组，保存的顺序如下:

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

初始化滤波器系数作为自适应滤波器的起始点。pState 指向一个长度为 numTaps+blockSize-1 样本数组, 其中 blockSize 是处理的输入样本数量，传入函数*csi_lms_norm_q15()* 。

第七章　插值函数

这些函数处理数据的一维和二维插值。线性插值用在一维数据，双线性插值用在二维数据。

7.1　线性插值

7.1.1　函数

- *csky_linear_interp_f32* : 浮点线性插值处理函数
- *csky_linear_interp_q31* : Q31 线性插值处理函数
- *csky_linear_interp_q15* : Q15 线性插值处理函数
- *csky_linear_interp_q7* : Q7 线性插值处理函数

7.1.2　简要说明

线性插值是一种利用线性多项式的曲线拟合方法。线性插值在相邻样本之间画一条直线，并返回线上的适当的点。

为了得到线性插值函数的输出值 (y), 先要根据 (x) 得到线性插值的输入 x0, x1(输入值最接近的值) ，然后得到输出值 y0 和 y1(输出值最接近的值)

算法:

```
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
```

这组函数实现了 Q7，Q15，Q31 和浮点数据类型的线性插值处理。函数一次操作一个单独的数据，并且每次调用返回一个单独的处理结果。S 指向线性插值函数数据结构体实例。x 是输入样本值，函数返回输出值。

如果 x 低于表中边界，线性插值返回表中第一个值，如果 x 是超过最大范围，则返回表中最后一个值。

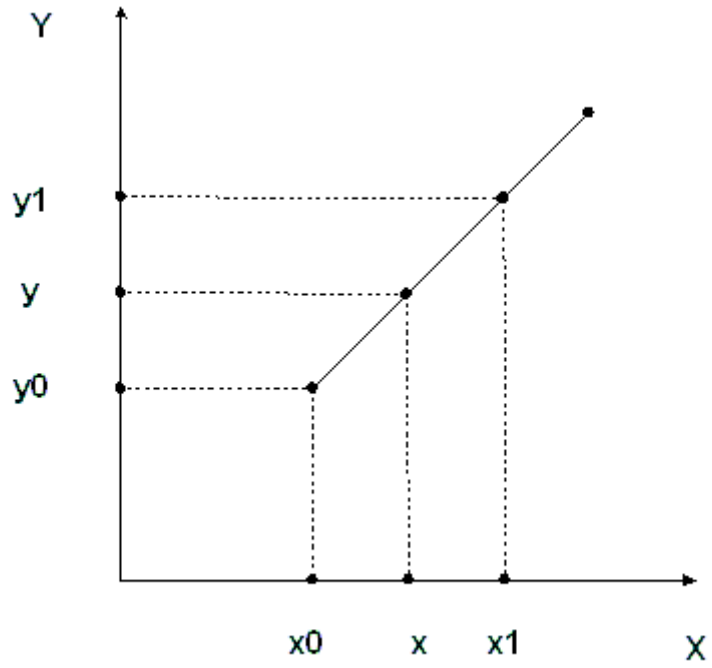


图 7.1: Linear interpolation

7.1.3 Function Docmentation

7.1.3.1 csky_linear_interp_f32

```
float32_t csky_linear_interp_f32 (csky_linear_interp_instance_f32 *S, float32_t x)
```

参数:

*S: 浮点线性插值结构体实例

x: 需要处理的输入样本

返回值:

输出结果

7.1.3.2 csky_linear_interp_q31

```
q31_t  csky_linear_interp_q31 (q31_t *pYData, q31_t x, uint32_t nValues)
```

参数:

*pYData: 浮点线性插值结构体实例

x: 需要处理的输入样本

nValues: 表中值的数量

返回值:

输出结果

简要说明:

输入样本 x 是 12.20 格式，包括了 12 位的表索引和 20 位的小数部分。函数支持的最大表大小是 2^12.

7.1.3.3 csky_linear_interp_q15

```
q15_t  csky_linear_interp_q15 (q15_t *pYData, q31_t x, uint32_t nValues)
```

参数:

*pYData: 浮点线性插值结构体实例
x: 需要处理的输入样本
nValues: 表中值的数量

返回值:

输出结果

简要说明:

输入样本 x 是 12.20 格式，包括了 12 位的表索引和 20 位的小数部分。函数支持的最大表大小是 2^12.

7.1.3.4 csky_linear_interp_q7

```
q7_t  csky_linear_interp_q7 (q7_t *pYData, q31_t x, uint32_t nValues)
```

参数:

*pYData: 浮点线性插值结构体实例
x: 需要处理的输入样本
nValues: 表中值的数量

返回值:

输出结果

简要说明:

输入样本 x 是 12.20 格式，包括了 12 位的表索引和 20 位的小数部分。函数支持的最大表大小是 2^12.

7.2 双线性插值

7.2.1 函数

- *csky_bilinear_interp_f32* : 浮点双线性插值
- *csky_bilinear_interp_q31* : Q15 双线性插值
- *csky_bilinear_interp_q15* : Q31 双线性插值
- *csky_bilinear_interp_q7* : Q7 双线性插值

7.2.2 简要说明

双线性插值是线性插值在二维网格上的扩展。底层函数 f(x, y) 获取一个规则网格，然后插值处理决定值落在网格中的哪个点。双线性插值等价于两次线性插值，首先是 x 轴，然后是 y 轴。双线性插值常用在图像处理的缩放图像上。CSI DSP 库提供 Q7，Q15，Q31 和浮点数据类型的双线性插值。

算法

双线性插值函数使用的结构体实例描述了一个二维数据表。对于浮点来说，结构体实例定义如下：

```
typedef struct
{
  uint16_t numRows;
  uint16_t numCols;
  float32_t *pData;
} csky_bilinear_interp_instance_f32;
```

其中 numRows 指定表的行数，numCols 指定表的列数，pData 指向一个大小为 numRows * numCols 数组. 数据表 pTable 按行排列，按整数值索引。就是说，表元素 (x,y) 落在 pTable[x + y * numCols] ，其中 x 和 y 是整数.

令 (x, y) 指定要插值的点，则定义：

```
XF = floor(x)
YF = floor(y)
```

插值的输出点计算如下：

```
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)
```

注意，坐标 (x, y) 包括整数和小数部分。整数部分指定使用表哪部分，小数部分控制插值处理

如果 (x,y) 超出了表的边界，双线性插值结果返回 0。

7.2.3 Function Docmentation

7.2.3.1 csky_bilinear_interp_f32

```
float32_t csky_bilinear_interp_f32 (const csky_bilinear_interp_instance_f32 *S,␣
↪float32_t X, float32_t Y)
```

参数:

*S: 指向插值结构体实例

X: 插值坐标

Y: 插值坐标

返回值:

插值结果

7.2.3.2 csky_bilinear_interp_q31

```
q31_t  csky_bilinear_interp_q31 (csky_bilinear_interp_instance_q31 *S, q31_t X, q31_t␣
↪Y)
```

参数:

*pYData: 指向插值结构体实例

X: 12.20 格式插值坐标

Y: 12.20 格式插值坐标

返回值:

插值结果

7.2.3.3 csky_bilinear_interp_q15

```
q15_t  csky_bilinear_interp_q15 (csky_bilinear_interp_instance_q15 *S, q31_t X, q31_t␣
↪Y)
```

参数:

*pYData: 指向插值结构体实例

X: 12.20 格式插值坐标

Y: 12.20 格式插值坐标

返回值:

插值结果

7.2.3.4 csky_bilinear_interp_q7

```
q7_t  csky_bilinear_interp_q7 (csky_bilinear_interp_instance_q7 *S, q31_t X, q31_t Y)
```

参数:

`*pYData`: 指向插值结构体实例

`X`: 12.20 格式插值坐标

`Y`: 12.20 格式插值坐标

返回值:

插值结果

第八章　矩阵函数

简要说明

这组函数提供了基本的矩阵操作。函数以矩阵数据结构为操作描述矩阵。比如，浮点矩阵数据结构体的类型定义如下：

```
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;
```

Q15 和 Q31 矩阵的数据类型也类似。

矩阵结构体指定了矩阵的大小，也指定了矩阵的数据数组。数组的大小是 numRows X numCols，数据按行顺序排列。就是说，矩阵的元素 (i, j) 保存在:

```
pData[i*numCols + j]
```

初始化函数

每种矩阵数据结构体类型都有一个相应的初始化函数。初始化函数设置数据结构内部的数值。函数 csi_mat_init_f32(), csi_mat_init_q31() 和 csi_mat_init_q15() 分别对应浮点，Q31 和 Q15 类型。

是否使用初始化函数是可选的。但是如果使用了初始化函数，则结构体实例不能放在常量数据段。想要将结构体实例放在常量数据段，则需要手动初始化数据结构。比如:

```
csi_matrix_instance_f32 S = {nRows, nColumns, pData};
csi_matrix_instance_q31 S = {nRows, nColumns, pData};
csi_matrix_instance_q15 S = {nRows, nColumns, pData};
```

其中 nRows 指定行数, nColumns 指定列数, pData 指向数据数组。

8.1 矩阵加法

8.1.1 函数

- *csi_mat_init_f32* : 浮点矩阵初始化
- *csi_mat_init_q31* : Q31 矩阵初始化
- *csi_mat_init_q15* : Q15 矩阵初始化

8.1.2 简要说明

初始化矩阵数据结构. 函数设置矩阵结构体的 numRows, numCols, 和 pData 字段

8.1.3 函数说明

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)
```

参数:

*S: 指向一个矩阵结构体实例
nRows: 矩阵的行数
nColumns: 矩阵的列数
*pData: 指向矩阵数据数组

返回值:

无

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)
```

参数:

*S: 指向一个矩阵结构体实例
nRows: 矩阵的行数
nColumns: 矩阵的列数
*pData: 指向矩阵数据数组

返回值:

无

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)
```

参数:

*S: 指向一个矩阵结构体实例

nRows: 矩阵的行数

nColumns: 矩阵的列数

*pData: 指向矩阵数据数组

返回值:

无

8.2 矩阵加法

8.2.1 函数

- *csi_mat_add_f32* : 浮点矩阵加法
- *csi_mat_add_q31* : Q31 矩阵加法
- *csi_mat_add_q15* : Q15 矩阵加法

8.2.2 简要说明

两个矩阵相加.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{bmatrix}$$

图 8.1: 两个 3 x 3 矩阵相加

8.2.3 函数说明

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)
```

参数:

`*pSrcA`: 指向第一个输入矩阵结构
`*pSrcB`: 指向第二个输入矩阵结构
`*pDst`: 指向输出矩阵结构

返回值:

无

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)
```

参数:

*pSrcA: 指向第一个输入矩阵结构

*pSrcB: 指向第二个输入矩阵结构

*pDst: 指向输出矩阵结构

返回值:

无

缩放和溢出行为:

函数使用饱和计算。超过 Q31 最大范围 [0x80000000 0x7FFFFFFF] 的结果会被饱和。

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)
```

参数:

*pSrcA: 指向第一个输入矩阵结构

*pSrcB: 指向第二个输入矩阵结构

*pDst: 指向输出矩阵结构

返回值:

无

缩放和溢出行为:

函数使用饱和计算。超过 Q15 最大范围 [0x8000 0x7FFF] 的结果会被饱和。

8.3 复数矩阵乘法

8.3.1 函数

- *csi_mat_cmplx_mult_f32* : 浮点复数矩阵乘法
- *csi_mat_cmplx_mult_q31* : Q31 复数矩阵乘法
- *csi_mat_cmplx_mult_q15* : Q15 复数矩阵乘法

8.3.2 简要说明

只有当第一个矩阵的列数和第二个矩阵的行数相等的时候，才可以做复数矩阵乘法。M x N 矩阵和 N x P 矩阵相乘的结果是一个 M x P 矩阵。当使能矩阵大小检查，函数会检查:

1. pSrcA 和 pSrcB 的内部尺寸是否相等
2. 输出向量的尺寸是否跟 pSrcA 和 pSrcB 的计算结果相等

8.3.3 函数说明

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)
```

参数:

`*pSrcA`: 指向第一个输入矩阵结构

`*pSrcB`: 指向第二个输入矩阵结构

`*pDst`: 指向输出矩阵结构

返回值:

无

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)
```

参数:

`*pSrcA`: 指向第一个输入矩阵结构

`*pSrcB`: 指向第二个输入矩阵结构

`*pDst`: 指向输出矩阵结构

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。累加器用 2.62 格式维持了中间结果的所有精度，但是只有一个保护位。累加时候没有饱和计算，因此累加器溢出会扭曲结果，需要缩小输入信号来防止中间结果溢出。因为最多会有 numColsA 个加法进位，所以需要缩小 log2(numColsA) 位来防止溢出。2.62 格式的累加器右移 31 位，然后饱和生成 1.31 格式的最终结果。

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)
```

参数:

`*pSrcA`: 指向第一个输入矩阵结构

`*pSrcB`: 指向第二个输入矩阵结构

`*pDst`: 指向输出矩阵结构

`*pScratch`: 指向保存中间结果的临时数组

返回值:

无

最佳性能的条件:

输入，输出，和缓存都需要 32 位对齐

限制:

如果芯片不支持分对齐访问，则定义宏 UNALIGNED_SUPPORT_DISABLE 同时，输入，输出，临时 buffer，都应该是 32 位对齐。

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入都是 1.15 格式的，乘法生成 2.30 结果。2.30 格式的中间结果在 34.30 格式的 64 位累加器累加。由于有 33 个保护位，所以不会有溢出风险。34.30 的结果丢弃低 15 位截断为 34.15 格式，然后饱和成为 1.15 格式的结果。

函数*csi_mat_mult_fast_q15()* 是这个函数的一个快速版本，但是丢失更多的精度

8.4 矩阵求逆

8.4.1 函数

- *csi_mat_inverse_f32* : 单精度矩阵求逆.
- *csi_mat_inverse_f64* : 双精度矩阵求逆.

8.4.2 简要说明

计算矩阵的逆矩阵，该算法不支持 I805。仅当输入矩阵是方阵且非奇异（行列式非零）时才定义逆矩阵。该函数检查输入和输出矩阵是否为正方形且大小相同。矩阵求逆对数值敏感，CSI DSP 库仅支持浮点矩阵的矩阵求逆。

算法 Gauss-Jordan 方法用于求逆。该算法执行一系列基本的行操作，直到将输入矩阵简化为单位矩阵。将相同的基本行操作序列应用于单位矩阵会产生逆矩阵。如果输入矩阵是奇异的，则算法终止并返回错误状态 CSKY_MATH_SINGULAR。

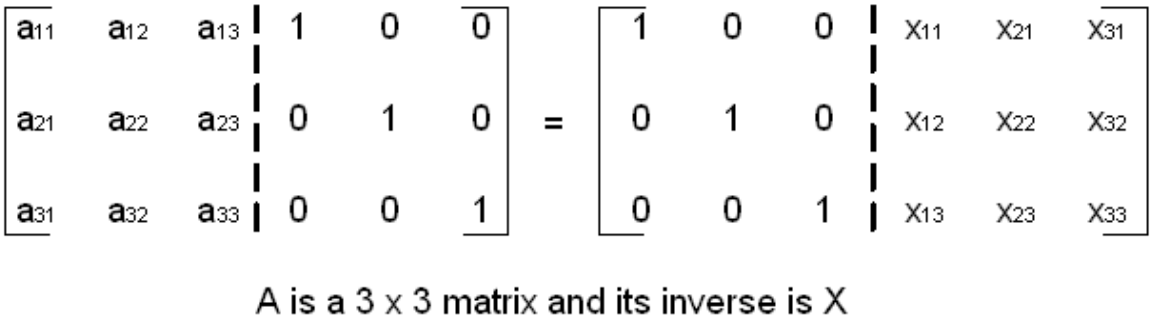


图 8.2: 使用 Gauss-Jordan 方法对 3x3 矩阵求逆

8.4.3 函数说明

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)
```

参数:

`*pSrc`: 指向输入矩阵结构.

`*pDst`: 指向输出矩阵结构.

返回值:

无

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)
```

参数:

*pSrc: 指向输入矩阵结构.

*pDst: 指向输入矩阵结构.

Returns:

无

8.5 矩阵乘法

8.5.1 函数

- *csi_mat_mult_f32* : 浮点矩阵乘法
- *csi_mat_mult_q31* : Q31 矩阵乘法
- *csi_mat_mult_q15* : Q15 矩阵乘法
- *csi_mat_mult_fast_q31* : Q31 矩阵乘法
- *csi_mat_mult_fast_q15* : Q15 矩阵乘法

8.5.2 简要说明

两个矩阵相乘，快速版本不支持 I805/C860。

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}xb_{11}+a_{12}xb_{21}+a_{13}xb_{31} & a_{11}xb_{12}+a_{12}xb_{22}+a_{13}xb_{32} & a_{11}xb_{13}+a_{12}xb_{23}+a_{13}xb_{33} \\ a_{21}xb_{11}+a_{22}xb_{21}+a_{23}xb_{31} & a_{21}xb_{12}+a_{22}xb_{22}+a_{23}xb_{32} & a_{21}xb_{13}+a_{22}xb_{23}+a_{23}xb_{33} \\ a_{31}xb_{11}+a_{32}xb_{21}+a_{33}xb_{31} & a_{31}xb_{12}+a_{32}xb_{22}+a_{33}xb_{32} & a_{31}xb_{13}+a_{32}xb_{23}+a_{33}xb_{33} \end{bmatrix}$$

图 8.3: 两个 3 x 3 矩阵相乘

只有第一个矩阵的列数和第二个矩阵的行数相等的时候，才可以做矩阵乘法。M x N 矩阵和 N x P 矩阵相乘的结果是一个 M x P 矩阵。

8.5.3 函数说明

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)
```

参数:

`*pSrcA`: 指向第一个输入矩阵结构

`*pSrcB`: 指向第二个输入矩阵结构

`*pDst`: 指向输出矩阵结构

返回值:

无

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)
```

参数:

*pSrcA: 指向第一个输入矩阵结构

*pSrcB: 指向第二个输入矩阵结构

*pDst: 指向输出矩阵结构

返回值:

无

缩放和溢出行为:

函数实现使用了一个 64 位的内部累加器。累加器使用 2.62 格式，维持了中间乘法结果的所有精度，但是只提供了 1 个保护位。累加过程中没有饱和处理，累加器溢出会导致结果扭曲，因此必须缩小输入信号，防止溢出。因为最多会有 numColsA 个加法进位，所以输入矩阵需要缩小 log2(numColsA) 个位，来防止溢出。2.62 格式的累加器右移 31 位，饱和生成 1.31 的最后结果。

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)
```

参数:

*pSrcA: 指向第一个输入矩阵结构

*pSrcB: 指向第二个输入矩阵结构

*pDst: 指向输出矩阵结构

*pState: 指向保存中间结果的数组

返回值:

无

缩放和溢出行为:

函数实现使用了一个 64 位的内部累加器。输入都是 1.15 格式，相乘产生的结果是 2.30 格式。2.30 的中间结果在 34.30 格式的累加器累加。因为提供了 33 个保护位，所有不会有溢出风险。34.30 的结果丢弃低 15 位，截断为 34.15 格式，然后饱和成 1.15 格式的最后结果

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)
```

参数:

*pSrcA: 指向第一个输入矩阵结构

*pSrcB: 指向第二个输入矩阵结构

*pDst: 指向输出矩阵结构

返回值:

无

缩放和溢出行为:

这个快速版本和函数*csi_mat_mult_q31()* 的区别在于，快速版本使用了一个 32 位的累加器，而不是 64 位的累加器。每个 1.31 和 1.31 相乘的结果截断为 2.30 的结果。中间结果在 2.30 格式的 32 位寄存器累加。最后累加器饱和转换为 1.31 格式的结果。

快速版本跟标准版本有相同的溢出行为，但是由于截断了每次相乘结果的低 32 位，所以提供的更低的精度。为了防止溢出，必须缩小输入信号。因为最多会有 numColsA 个加法进位，所以输入矩阵需要缩小 log2(numColsA) 个位，来防止溢出。

函数*csi_mat_mult_q31()* 是这个函数的一个慢速版本，使用 64 位累加器提供更高的精度。

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)
```

参数:

*pSrcA: 指向第一个输入矩阵结构

*pSrcB: 指向第二个输入矩阵结构

*pDst: 指向输出矩阵结构

*pState: 指向保存中间结果的数组

返回值:

无

缩放和溢出行为:

这个快速版本和函数*csi_mat_mult_q15()* 的区别在于，快速版本使用了一个 32 位的累加器，而不是 64 位的累加器。每个 1.15 和 1.15 相乘的结果截断为 2.30 的结果。中间结果在 2.30 格式的 32 位寄存器累加。最后累加器饱和转换为 1.15 格式的结果。

快速版本跟标准版本有相同的溢出行为，但是由于截断了每次相乘结果的低 16 位，所以提供的更低的精度。为了防止溢出，必须缩小输入信号。因为最多会有 numColsA 个加法进位，所以输入矩阵需要缩小 log2(numColsA) 个位，来防止溢出。

函数*csi_mat_mult_q15()* 是这个函数的一个慢速版本，使用 64 位累加器提供更高的精度

8.6 矩阵缩放

8.6.1 函数

- *csi_mat_scale_f32* : 浮点矩阵缩放
- *csi_mat_scale_q31* : Q31 矩阵缩放
- *csi_mat_scale_q15* : Q15 矩阵缩放

8.6.2 简要说明

矩阵与标量相乘，矩阵中的每个元素都与一个标量相乘比如:

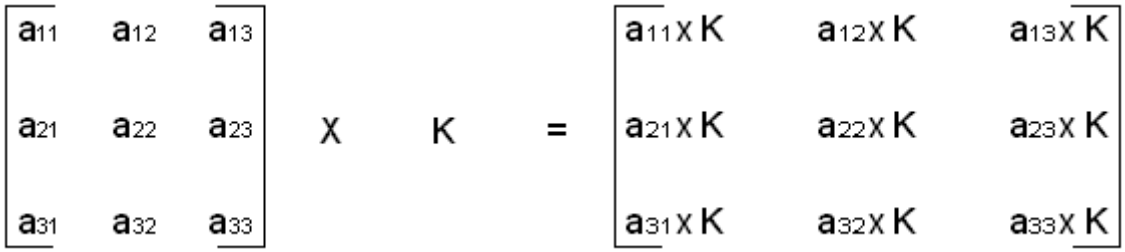


图 8.4: 3x3 矩阵缩放

Q15 和 Q31 的定点函数, scale 表示为一个小数乘法 scaleFract 和一个移位 shift. 移位让缩放操作的增益可以超过 1.0。定点数据的总的缩放因子是:

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

8.6.3 函数说明

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)
```

参数:

`*pSrc`: 指向输入矩阵结构体

`scale`: 缩放因子

`*pDst`: 指向输出矩阵结构体

返回值:

无

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)
```

参数:

*pSrc: 指向输入矩阵结构体

scaleFract: 缩放因子

shift: 结果的移位数量

*pDst: 指向输出矩阵结构体

返回值:

无

缩放和溢出行为:

输入数据 pSrc 和 scaleFract 都是 1.31 格式. 它们相乘的生成 2.62 的中间结果，然后移位和饱和成 1.31 格式

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)
```

参数:

*pSrc: 指向输入矩阵结构体

scaleFract: 缩放因子

shift: 结果的移位数量

*pDst: 指向输出矩阵结构体

返回值:

无

缩放和溢出行为:

输入数据 pSrc 和 scaleFract 都是 1.15 格式. 它们相乘的生成 2.30 的中间结果，然后移位和饱和成 1.15 格式

8.7　矩阵减法

8.7.1　函数

- *csi_mat_sub_f32* : 浮点矩阵相减
- *csi_mat_sub_q31* : Q31 矩阵相减
- *csi_mat_sub_q15* : Q15 矩阵相减

8.7.2　简要说明

两个矩阵相减

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} - \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}-b_{11} & a_{12}-b_{12} & a_{13}-b_{13} \\ a_{21}-b_{21} & a_{22}-b_{22} & a_{23}-b_{23} \\ a_{31}-b_{31} & a_{32}-b_{32} & a_{33}-b_{33} \end{bmatrix}$$

图 8.5: 两个 3 x 3 矩阵相减

8.7.3　函数说明

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)
```

参数:

`*pSrcA`: 指向第一个输入矩阵结构体
`*pSrcB`: 指向第二个输入矩阵结构体
`*pDst`: 指向输出矩阵结构体

返回值:

无

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)
```

参数:

*pSrcA: 指向第一个输入矩阵结构体

*pSrcB: 指向第二个输入矩阵结构体

*pDst: 指向输出矩阵结构体

返回值:

无

缩放和溢出行为:

函数使用饱和计算. 结果如果超出 Q31 的最大范围 [0x80000000 0x7FFFFFFF] 会被饱和.

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)
```

参数:

*pSrcA: 指向第一个输入矩阵结构体

*pSrcB: 指向第二个输入矩阵结构体

*pDst: 指向输出矩阵结构体

返回值:

无

缩放和溢出行为:

函数使用饱和计算. 结果如果超出 Q15 的最大范围 [0x8000 0x7FFF] 会被饱和.

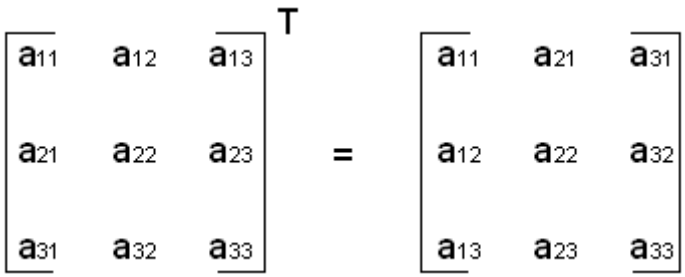
8.8 矩阵转置

8.8.1 函数

- *csi_mat_trans_f32* : 浮点矩阵转置
- *csi_mat_trans_q31* : Q31 矩阵转置
- *csi_mat_trans_q15* : Q15 矩阵转置

8.8.2 简要说明

转置一个矩阵. 转置一个 M x N 就是让它按中心对角线翻转出一个 N x M 矩阵



转置一个 3 x 3 矩阵

8.8.3 函数说明

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)
```

参数:

`*pSrc`: 指向输入矩阵

`*pDst`: 指向输出矩阵

返回值:

无

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)
```

参数:

*pSrc: 指向输入矩阵

*pDst: 指向输出矩阵

返回值:

无

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)
```

参数:

*pSrc: 指向输入矩阵

*pDst: 指向输出矩阵

返回值:

无

第九章　统计函数

9.1　最大值

9.1.1　函数

- *csi_max_f32* : 浮点数组中的最大值
- *csi_max_q31* : Q31 数组中的最大值
- *csi_max_q15* : Q15 数组中的最大值
- *csi_max_q7* : Q7 数组中的最大值

9.1.2　简要说明

取一个数组中的最大值。函数返回最大值和最大值在数组中的位置。为浮点, Q31, Q15 和 Q7 数据类型分别提供不同的函数。

9.1.3　函数说明

9.1.3.1　csi_max_f32

```
void csi_max_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult, uint32_t␣
↪*pIndex)
```

参数:

`*pSrc`: 指向输入数组

`blockSize`: 输入数组的长度

`*pResult`: 返回的最大值

`*pIndex`: 返回的最大值的索引

返回值:

无

9.1.3.2 csi_max_q31

```
void csi_max_q31 (q31_t *pSrc, uint32_t blockSize, q31_t *pResult, uint32_t *pIndex)
```

参数:

*pSrc: 指向输入数组
blockSize: 输入数组的长度
*pResult: 返回的最大值
*pIndex: 返回的最大值的索引

返回值:

无

9.1.3.3 csi_max_q15

```
void csi_max_q15 (q15_t *pSrc, uint32_t blockSize, q15_t *pResult, uint16_t *pIndex)
```

参数:

*pSrc: 指向输入数组
blockSize: 输入数组的长度
*pResult: 返回的最大值
*pIndex: 返回的最大值的索引

返回值:

无

9.1.3.4 csi_max_q7

```
void csi_max_q7 (q7_t *pSrc, uint32_t blockSize, q7_t *pResult, uint16_t *pIndex)
```

参数:

*pSrc: 指向输入数组
blockSize: 输入数组的长度
*pResult: 返回的最大值
*pIndex: 返回的最大值的索引

返回值:

无

9.2 最小值

9.2.1 函数

- *csi_min_f32* : 浮点数组的最小值
- *csi_min_q31* : Q31 数组的最小值
- *csi_min_q15* : Q15 数组的最小值
- *csi_min_q7* : Q7 数组的最小值

9.2.2 简要说明

计算数组数据中的最小值. 函数返回最小值和最小值在数组中的位置。为浮点,Q31，Q15，Q7 分别提供了不同的函数。

9.2.3 函数说明

9.2.3.1 csi_min_f32

```
void csi_min_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult, uint32_t␣
↪*pIndex)
```

参数:

`*pSrc`: 指向输入数组
`blockSize`: 输入数组的长度
`*pResult`: 返回的最大值
`*pIndex`: 返回的最大值的索引

返回值:

无

9.2.3.2 csi_min_q31

```
void csi_min_q31 (q31_t *pSrc, uint32_t blockSize, q31_t *pResult, uint32_t *pIndex)
```

参数:

`*pSrc`: 指向输入数组
`blockSize`: 输入数组的长度
`*pResult`: 返回的最大值
`*pIndex`: 返回的最大值的索引

返回值:

无

9.2.3.3 csi_min_q15

```
void csi_min_q15 (q15_t *pSrc, uint32_t blockSize, q15_t *pResult, uint16_t *pIndex)
```

参数:

*pSrc: 指向输入数组
blockSize: 输入数组的长度
*pResult: 返回的最大值
*pIndex: 返回的最大值的索引

返回值:

无

9.2.3.4 csi_min_q7

```
void csi_min_q7 (q7_t *pSrc, uint32_t blockSize, q7_t *pResult, uint16_t *pIndex)
```

参数:

*pSrc: 指向输入数组
blockSize: 输入数组的长度
*pResult: 返回的最大值
*pIndex: 返回的最大值的索引

返回值:

无

9.3 平均值

9.3.1 函数

- *csi_mean_f32* : 浮点数组的平均值
- *csi_mean_q31* : Q31 数组的平均值
- *csi_mean_q15* : Q15 数组的平均值
- *csi_mean_q7* : Q7 数组的平均值

9.3.2 简要说明

计算输入数组的平均值。数组中的所有元素的平均值。使用下面的算法:

```
Result = (pSrc[0] + pSrc[1] + pSrc[2] + ... + pSrc[blockSize-1]) / blockSize;
```

为浮点,Q31，Q15 和 Q7 数据类型分别提供了不同的函数

9.3.3 函数说明

9.3.3.1 csi_mean_f32

```
void csi_mean_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

参数:

`*pSrc`: 指向输入数组
`blockSize`: 输入数组的长度
`*pResult`: 返回的平均值

返回值:

无

9.3.3.2 csi_mean_q31

```
void csi_mean_q31 (q31_t *pSrc, uint32_t blockSize, q31_t *pResult)
```

参数:

`*pSrc`: 指向输入数组
`blockSize`: 输入数组的长度
`*pResult`: 返回的平均值

返回值:

无

缩放和溢出行为:

函数实现使用了一个 64 位内部累加器。输入表示为 1.31 格式，累加器的格式是 33.31。因此可以保留所有的中间结果精度，不会有溢出风险。最后，结果截断为 1.31 格式。

9.3.3.3 csi_mean_q15

```
void csi_mean_q15 (q15_t *pSrc, uint32_t blockSize, q15_t *pResult)
```

参数:

`*pSrc`: 指向输入数组

`blockSize`: 输入数组的长度

`*pResult`: 返回的平均值

返回值:

无

缩放和溢出行为:

函数实现使用一个内部 32 位累加器。输入表示为 1.15 格式，32 位累加器用 17.15 格式。因此可以保留所有的精度，不会有溢出风险。最后，累加器饱和截断生成 1.15 格式的结果。

9.3.3.4 csi_mean_q7

```
void csi_mean_q7 (q7_t *pSrc, uint32_t blockSize, q7_t *pResult)
```

参数:

`*pSrc`: 指向输入数组

`blockSize`: 输入数组的长度

`*pResult`: 返回的平均值

返回值:

无

缩放和溢出行为:

函数实现使用了一个 32 位内部累加器。输入表示为 1.7 格式，累加器的格式是 25.7。因此可以保留所有的中间结果精度，不会有溢出风险。最后，结果截断为 1.7 格式。

9.4 平方和

9.4.1 函数

- *csi_power_int32* : 32 位整数所有元素的平方之和
- *csi_power_f32* : 浮点数组中所有元素的平方和
- *csi_power_q31* : Q31 数组中所有元素的平方和
- *csi_power_q15* : Q15 数组中所有元素的平方和
- *csi_power_q7* : Q7 数组中所有元素的平方和

9.4.2 简要说明

计算输入数组中元素的平方和. 其中 E906/E907 不支持 power_int32，算法如下:

```
Result = pSrc[0] * pSrc[0] + pSrc[1] * pSrc[1] + pSrc[2] * pSrc[2] + ... +␣
↪pSrc[blockSize-1] * pSrc[blockSize-1];
```

为浮点,Q31，Q15，Q7 分别提供了不同的函数。

9.4.3 函数说明

9.4.3.1 csi_power_int32

```
void csi_power_int32 (int32 *pSrc, uint32_t blockSize, q63_t *pResult)
```

参数:

`*pSrc`: 指向输入数组
`blockSize`: 输入数组的长度
`*pResult`: 返回的平方和

返回值:

无

缩放和溢出行为:

这个函数的输入为 32 位整数，元素平方后扩展为 64 位整数，相加使用了 64 位累加器。相加结果会被饱和，最后的结果会在 [0x0, 0x7fffffffffffff] 之间。因此，在使用时，需要注意溢出。

9.4.3.2 csi_power_f32

```
void csi_power_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

参数:

*pSrc: 指向输入数组

blockSize: 输入数组的长度

*pResult: 返回的平方和

返回值:

无

9.4.3.3 csi_power_q31

```
void csi_power_q31 (q31_t *pSrc, uint32_t blockSize, q31_t *pResult)
```

参数:

*pSrc: 指向输入数组

blockSize: 输入数组的长度

*pResult: 返回的平方和

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入数据表示为 1.31 格式。中间乘法生成 2.62 格式，然后截断为 2.48 格式的结果，结果在 16.48 格式的 64 位累加器累加。因为有 15 位保护位，所以不会有溢出的风险。同时还可以保存所有的中间乘法结果的精度. 最后，返回结果是 16.48 格式.

9.4.3.4 csi_power_q15

```
void csi_power_q15 (q15_t *pSrc, uint32_t blockSize, q15_t *pResult)
```

参数:

*pSrc: 指向输入数组

blockSize: 输入数组的长度

*pResult: 返回的平方和

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入数据表示为 1.15 格式。中间乘法生成 2.30 格式的结果，结果在 34.30 格式的 64 位累加器累加。因为有 33 位保护位，所以不会有溢出的风险。同时还可以保存所有的中间乘法结果的精度。最后，返回结果是 34.30 格式.

9.4.3.5 csi_power_q7

```
void csi_power_q7 (q7_t *pSrc, uint32_t blockSize, q7_t *pResult)
```

参数:

`*pSrc`: 指向输入数组

`blockSize`: 输入数组的长度

`*pResult`: 返回的平方和

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 32 位累加器。输入数据表示为 1.7 格式。中间乘法生成 2.14 格式的结果，结果在 18.14 格式的 64 位累加器累加。因为有 17 位保护位，所以不会有溢出的风险。同时还可以保存所有的中间乘法结果的精度最后，返回结果是 18.14 格式.

9.5 均方根 (RMS)

9.5.1 函数

- *csi_rms_f32* : 浮点数组元素的均方根
- *csi_rms_q31* : Q31 数组元素的均方根
- *csi_rms_q15* : Q15 数组元素的均方根

9.5.2 简要说明

计算输入数组中所有元素的均方根. 算法如下:

```
Result = sqrt(((pSrc[0] * pSrc[0] + pSrc[1] * pSrc[1] + ... + pSrc[blockSize-
↪1] * pSrc[blockSize-1]) / blockSize));
```

为浮点,Q31，Q15 分别提供了不同的函数。

9.5.3 函数说明

9.5.3.1 csi_rms_f32

```
void csi_rms_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

参数:

`*pSrc`: 指向输入数组
`blockSize`: 输入数组的长度
`*pResult`: 返回的均方根结果

返回值:

无

9.5.3.2 csi_rms_q31

```
void csi_rms_q31 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

参数:

`*pSrc`: 指向输入数组
`blockSize`: 输入数组的长度
`*pResult`: 返回的均方根结果

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入数据表示为 1.31 格式。中间乘法生成 2.62 格式的结果，结果在 2.62 格式的 64 位累加器累加。因为只有一个保护位，所以需要缩小输入信号来防止溢出。因为最多会有 blockSize 个加法进位，所以需要缩小 log2(blockSize) 才可以保证没有溢出。最后，2.62 格式右移 31 位，然后饱和生成 1.15 格式的结果。

9.5.3.3 csi_rms_q15

```
void csi_rms_q15 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

参数:

`*pSrc`: 指向输入数组

`blockSize`: 输入数组的长度

`*pResult`: 返回的均方根结果

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入数据表示为 1.15 格式。中间乘法生成 2.30 格式的结果，结果在 34.30 格式的 64 位累加器累加。因为有 33 位保护位，所以不会有溢出的风险。同时还可以保存所有的中间乘法结果的精度。最后，34.30 格式的丢弃低 15 位截断为 34.15，然后饱和生成 1.15 格式的结果。

9.6 标准偏差

9.6.1 函数

- *csi_std_f32* : 浮点数组元素的标准偏差
- *csi_std_q31* : Q31 数组元素的标准偏差
- *csi_std_q15* : Q15 数组元素的标准偏差

9.6.2 简要说明

计算输入数组元素的标准偏差使用的算法如下:

```
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]
```

为浮点，Q31，Q15 数据类型分别提供不同的函数。

9.6.3 函数说明

9.6.3.1 csi_std_f32

```
void csi_std_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

参数:

`*pSrc`: 指向输入数组

`blockSize`: 输入数组的长度

`*pResult`: 返回的标准偏差

返回值:

无

9.6.3.2 csi_std_q31

```
void csi_std_q31 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

参数:

*pSrc: 指向输入数组

blockSize: 输入数组的长度

*pResult: 返回的标准偏差

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入数据表示为 1.31 格式，然后移位 8 位，生成 1.23 格式。中间乘法生成 2.46 格式的结果，结果在 18.46 格式的 64 位累加器累加。除法之后，中间结果应该是 18.46 格式。最后，18.46 格式右移 15 位，然后饱和生成 1.31 格式的结果。

9.6.3.3 csi_std_q15

```
void csi_std_q15 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

参数:

*pSrc: 指向输入数组

blockSize: 输入数组的长度

*pResult: 返回的标准偏差

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入数据表示为 1.15 格式。中间乘法生成 2.30 格式的结果，结果在 34.30 格式的 64 位累加器累加。因为有 33 位保护位，所以不会有溢出的风险。同时还可以保存所有的中间乘法结果的精度。最后，34.30 格式的丢弃低 15 位截断为 34.15，然后饱和生成 1.15 格式的结果。

9.7 方差

9.7.1 函数

- *csi_var_f32* : 浮点数组元素的方差
- *csi_var_q31* : Q31 数组元素的方差
- *csi_var_q15* : Q15 数组元素的方差

9.7.2 简要说明

计算输入数组元素的方差。使用的算法如下:

```
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]
```

为浮点，Q31 和 Q15 分别提供了不同的函数。

9.7.3 函数说明

9.7.3.1 csi_var_f32

```
void csi_var_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

参数:

`*pSrc`: 指向输入数组

`blockSize`: 输入数组的长度

`*pResult`: 返回的方差

返回值:

无

9.7.3.2 csi_var_q31

```
void csi_var_q31 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

参数:

*pSrc: 指向输入数组

blockSize: 输入数组的长度

*pResult: 返回的方差

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入数据表示为 1.31 格式，然后移位 8 位，生成 1.23 格式。中间乘法生成 2.46 格式的结果，结果在 18.46 格式的 64 位累加器累加。除法之后，中间结果应该是 18.46 格式。最后，18.46 格式右移 15 位，然后饱和生成 1.31 格式的结果。

9.7.3.3 csi_var_q15

```
void csi_var_q15 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)
```

参数:

*pSrc: 指向输入数组

blockSize: 输入数组的长度

*pResult: 返回的方差

返回值:

无

缩放和溢出行为:

函数实现使用了一个内部 64 位累加器。输入数据表示为 1.15 格式。中间乘法生成 2.30 格式的结果，结果在 34.30 格式的 64 位累加器累加。因为有 33 位保护位，所以不会有溢出的风险。同时还可以保存所有的中间乘法结果的精度。最后，34.30 格式的丢弃低 15 位截断为 34.15，然后饱和生成 1.15 格式的结果。

第十章　辅助函数

10.1　向量复制

10.1.1　函数

- *csi_copy_f32* : 复制浮点向量元素
- *csi_copy_q31* : 复制 Q31 向量元素
- *csi_copy_q15* : 复制 Q15 向量元素
- *csi_copy_q7* : 复制 Q7 向量元素

10.1.2　简要说明

把样本一个个从源向量复制到目的向量。

```
pDst[n] = pSrc[n];   0 <= n < blockSize.
```

为浮点，Q31，Q15 和 Q7 数据类型分别提供不同的函数。

10.1.3　函数说明

10.1.3.1　csi_copy_f32

```
void csi_copy_f32 (float32_t *pSrc, float32_t *pDst, uint32_t blockSize)
```

参数:

`*pSrc`: 指向输入向量

`*pDst`: 指向输出向量

`blockSize`: 输入向量的元素数量

返回值:

无

10.1.3.2 csi_copy_q31

```
void csi_copy_q31 (q31_t *pSrc, q31_t *pDst, uint32_t blockSize)
```

参数:

*pSrc: 指向输入向量
*pDst: 指向输出向量
blockSize: 输入向量的元素数量

返回值:

无

10.1.3.3 csi_copy_q15

```
void csi_copy_q15 (q15_t *pSrc, q15_t *pDst, uint32_t blockSize)
```

参数:

*pSrc: 指向输入向量
*pDst: 指向输出向量
blockSize: 输入向量的元素数量

返回值:

无

10.1.3.4 csi_copy_q7

```
void csi_copy_q7 (q7_t *pSrc, q7_t *pDst, uint32_t blockSize)
```

参数:

*pSrc: 指向输入向量
*pDst: 指向输出向量
blockSize: 输入向量的元素数量

返回值:

无

10.2 向量填充

10.2.1 函数

- *csi_fill_f32* : 用常量填充浮点向量
- *csi_fill_q31* : 用常量填充 Q31 向量
- *csi_fill_q15* : 用常量填充 Q31 向量
- *csi_fill_q7* : 用常量填充 Q7 向量 v

10.2.2 简要说明

用一个常量值填充目的向量.

```
pDst[n] = value;   0 <= n < blockSize.
```

为浮点，Q31，Q15 和 Q7 数据类型分别提供不同的函数。

10.2.3 函数说明

10.2.3.1 csi_fill_f32

```
void csi_fill_f32 (float32_t value, float32_t *pDst, uint32_t blockSize)
```

参数:

`value`: 用来填充的值
`*pDst`: 指向输出向量
`blockSize`: 输出向量的元素数量

返回值:

无

10.2.3.2 csi_fill_q31

```
void csi_fill_q31 (q31_t value, q31_t *pDst, uint32_t blockSize)
```

参数:

`value`: 用来填充的值
`*pDst`: 指向输出向量
`blockSize`: 输出向量的元素数量

返回值:

无

10.2.3.3 csi_fill_q15

```
void csi_fill_q15 (q15_t value, q15_t *pDst, uint32_t blockSize)
```

参数:

value: 用来填充的值
*pDst: 指向输出向量
blockSize: 输出向量的元素数量

返回值:

无

10.2.3.4 csi_fill_q7

```
void csi_fill_q7 (q7_t value, q7_t *pDst, uint32_t blockSize)
```

参数:

value: 用来填充的值
*pDst: 指向输出向量
blockSize: 输出向量的元素数量

返回值:

无

10.3 转换浮点到定点

10.3.1 函数

- *csi_float_to_q15* : 将浮点向量的元素转换到 Q15 向量
- *csi_float_to_q31* : 转换浮点向量元素到 Q31 向量
- *csi_float_to_q7* : 转换浮点向量元素到 Q7 向量

10.3.2 函数说明

10.3.2.1 csi_float_to_q15

```
void csi_float_to_q15 (float32_t *pSrc, q15_t *pDst, uint32_t blockSize)
```

参数:

`*pSrc`: 指向输入的向量

`*pDst`: 指向输出的向量

`blockSize`: 输入向量的元素数量

返回值:

无

简要说明:

转换过程用的公式是:

```
pDst[n] = (q15_t)(pSrc[n] * 32768);   0 <= n < blockSize.
```

缩放和溢出时的行为:

函数使用饱和算法. 结果如果超出 Q15 的范围 [0x8000 0x7FFF] 会被饱和

10.3.2.2 csi_float_to_q31

```
void csi_float_to_q31 (float32_t *pSrc, q31_t *pDst, uint32_t blockSize)
```

参数:

`*pSrc`: 指向输入的向量

`*pDst`: 指向输出的向量

`blockSize`: 输入向量的元素数量

返回值:

无

简要说明:

转换过程的公式如下:

```
pDst[n] = (q31_t)(pSrc[n] * 2147483648);   0 <= n < blockSize.
```

缩放和溢出时的行为:

函数使用饱和算法. 结果如果超出 Q31 的最大范围 [0x80000000 0x7FFFFFFF] 则会被饱和

10.3.2.3 csi_float_to_q7

```
void csi_float_to_q7 (float32_t *pSrc, q7_t *pDst, uint32_t blockSize)
```

参数:

`*pSrc`: 指向输入的向量

`*pDst`: 指向输出的向量

`blockSize`: 输入向量的元素数量

返回值:

无

简要说明:

转换处理的公式如下:

```
pDst[n] = (q7_t)(pSrc[n] * 128);   0 <= n < blockSize.
```

缩放和溢出时的行为:

函数使用饱和算法. 结果如果超出 Q31 的最大范围 [0x80 0x7F] 则会被饱和.

10.4 转换 Q15 的值

10.4.1 函数

- *csi_q15_to_float* : 转换 Q15 向量到浮点向量
- *csi_q15_to_q31* : 转换 Q15 向量元素到 Q31 向量
- *csi_q15_to_q7* : 转换 Q15 向量元素到 Q7 向量

10.4.2 函数说明

10.4.2.1 csi_q15_to_float

```
void csi_q15_to_float (q15_t *pSrc, float32_t *pDst, uint32_t blockSize)
```

参数:

`*pSrc`: 指向输入的向量

`*pDst`: 指向输出的向量

`blockSize`: 输入向量的元素数量

返回值:

无

简要说明:

转换过程的公式如下:

```
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)
```

参数:

`*pSrc`: 指向输入的向量

`*pDst`: 指向输出的向量

`blockSize`: 输入向量的元素数量

返回值:

无

简要说明:

转换过程的公式如下:

```
pDst[n] = (q31_t) pSrc[n] << 16;   0 <= n < blockSize.
```

10.4.2.3 csi_q15_to_q7

```
void csi_q15_to_q7 (q15_t *pSrc, q7_t *pDst, uint32_t blockSize)
```

参数:

*pSrc: 指向输入的向量

*pDst: 指向输出的向量

blockSize: 输入向量的元素数量

返回值:

无

简要说明:

转换过程的公式如下:

```
pDst[n] = (q7_t) pSrc[n] >> 8;   0 <= n < blockSize.
```

10.5 转换 Q31 的值

10.5.1 函数

- *csi_q31_to_float* : 转换 Q31 向量的元素到浮点向量
- *csi_q31_to_q15* : 转换 Q31 向量元素到 Q15 向量
- *csi_q31_to_q7* : 转换 Q31 向量元素到 Q7 向量

10.5.2 函数说明

10.5.2.1 csi_q31_to_float

```
void csi_q31_to_float (q31_t *pSrc, float32_t *pDst, uint32_t blockSize)
```

参数:

`*pSrc`: 指向输入的向量

`*pDst`: 指向输出的向量

`blockSize`: 输入向量的元素数量

返回值:

无

简要说明:

转换过程的公式如下:

```
pDst[n] = (float32_t) pSrc[n] / 2147483648;   0 <= n < blockSize.
```

10.5.2.2 csi_q31_to_q15

```
void csi_q31_to_q15 (q31_t *pSrc, q15_t *pDst, uint32_t blockSize)
```

参数:

`*pSrc`: 指向输入的向量

`*pDst`: 指向输出的向量

`blockSize`: 输入向量的元素数量

返回值:

无

简要说明:

转换过程的公式如下:

```
pDst[n] = (q15_t) pSrc[n] >> 16;   0 <= n < blockSize.
```

10.5.2.3 csi_q31_to_q7

```
void csi_q31_to_q7 (q31_t *pSrc, q7_t *pDst, uint32_t blockSize)
```

参数:

*pSrc: 指向输入的向量

*pDst: 指向输出的向量

blockSize: 输入向量的元素数量

返回值:

无

简要说明:

转换过程的公式如下:

```
pDst[n] = (q7_t) pSrc[n] >> 24;   0 <= n < blockSize.
```

10.6 转换 Q7 的值

10.6.1 函数

- *csi_q7_to_float* : 转换 Q7 向量元素到浮点向量
- *csi_q7_to_q15* : 转换 Q7 向量元素到 Q15 向量
- *csi_q7_to_q31* : 转换 Q7 向量元素到 Q31 向量

10.6.2 函数说明

10.6.2.1 csi_q7_to_float

```
void csi_q7_to_float (q7_t *pSrc, float32_t *pDst, uint32_t blockSize)
```

参数:

`*pSrc`: 指向输入的向量
`*pDst`: 指向输出的向量
`blockSize`: 输入向量的元素数量

返回值:

无

简要说明:

转换过程的公式如下:

```
pDst[n] = (float32_t) pSrc[n] / 128;   0 <= n < blockSize.
```

10.6.2.2 csi_q7_to_q15

```
void csi_q7_to_q15 (q7_t *pSrc, q15_t *pDst, uint32_t blockSize)
```

参数:

`*pSrc`: 指向输入的向量
`*pDst`: 指向输出的向量
`blockSize`: 输入向量的元素数量

返回值:

无

简要说明:

转换过程的公式如下:

```
pDst[n] = (q15_t) pSrc[n] << 8;   0 <= n < blockSize.
```

10.6.2.3　csi_q7_to_q31

```
void csi_q7_to_q31 (q7_t *pSrc, q31_t *pDst, uint32_t blockSize)
```

参数:

`*pSrc`: 指向输入的向量

`*pDst`: 指向输出的向量

`blockSize`: 输入向量的元素数量

返回值:

无

简要说明:

转换过程的公式如下:

```
pDst[n] = (q31_t) pSrc[n] << 24;   0 <= n < blockSize.
```

第十一章　变换函数

11.1　复数 FFT 函数

11.1.1　函数

- *csi_cfft_f32* : 浮点复数 FFT 处理函数
- *csi_cfft_q15* : Q15 复数 FFT 处理函数
- *csi_cfft_q31* : Q31 复数 FFT 处理函数

11.1.2　简要说明

快速傅里叶变换 (FFT) 是一个离散傅里叶变换（DFT）的快速算法。FFT 比 DFT 快了几个数量级，特别是处理较长的序列。这节描述的算法是处理复数数据的，另外有一组函数专门用于处理实数序列。

为浮点，Q15 和 Q31 分别提供了不同的算法。

FFT 函数在原地操作。也就是说，保存输入数据的数组也会被用作保存对应的结果。输入数据是复数的，并且包括 2*fftLen 个交错的数据，如下所示：

```
{real[0], imag[0], real[1], imag[1],..}
```

FFT 结果也会包含在相同的数组，并且频域的值也会有一样的交错方式。

浮点

浮点复数 FFT 使用一种混合基算法。多个 radix-8 步骤跟单个的 radix-2 或者 radix-4 步骤一起处理。算法支持长度 [16, 32, 64, ⋯, 4096] ，并且每种长度分别使用不同的旋转因子表。

函数使用了标准 FFT 定义，当计算正向变换的时候，输出值可能增长 fftLen。逆变换包括一个缩放 1/fftLen 作为计算的一部分，这跟教科书中的逆 FFT 定义相符

预初始化数据结构提供了旋转因子和位翻转表，都定义在 csi_const_structs.h. 可以你的函数里面包含这个头文件，然后将其作为参数传给 csi_cfft_f32. 比如：

```
csi_cfft_f32(csi_cfft_sR_f32_len64, pSrc, 1, 1)
```

是带位翻转的计算 64 点的复数逆 FFT。数据结构被视作常量，不会在计算过程中改变。相同的数据结构可以在多个正逆变换中重复使用。

库的早期发布版本分别为浮点类型提供了 radix-2 和 radix-4 算法。这些函数现在不推荐使用。新函数相对更快，也更通用。

如下是一个初始化 csi_cfft_f32 函数常量的示例：

```
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;
    break;
  case 4096:
    S = &csi_cfft_sR_f32_len4096;
    break;
  }
```

Q15 和 Q31

定点复数 FFT 使用了一种混合基算法. 多个 radix-4 步骤和单个 radix-2 步骤一起处理。算法支持长度 [16, 32, 64, ⋯, 4096]，并且分别为不同的长度提供旋转因子表。

函数使用了标准 FFT 定义，当计算正向变换的时候，输出值可能会增长 fftLen 。逆变换包括了一个缩放 1/fftLen 作为计算的一部分，这跟教科书中的逆 FFT 定义相符。

与初始化的数据结构提供了旋转因子和位翻转表，定义在头文件 csi_const_structs.h. 可以你的函数里面包含这个头文件，然后将其作为参数传给 csi_cfft_q31. 比如：

```
csi_cfft_q31(csi_cfft_sR_q31_len64, pSrc, 1, 1)
```

是一个包括了位翻转的 64 点的复数逆 FFT。数据结构被视作常量，不会在计算过程中改变。相同的数据结构可以在多个正逆变换中重复使用。

库的早期发布版本分别为定点类型提供了 radix-2 和 radix-4 算法。这些函数现在不推荐使用。新函数相对更快，也更通用。

如下是一个初始化 csi_cfft_q31 函数常量的示例：

```
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;
    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 函数说明

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)
```

参数:

*S: 指向 CFFT 结构体实例

*p1: 指向复数数据缓存，大小是 2*fftLen. 原地处理

ifftFlag: 设置是正向 (ifftFlag=0) 还是逆向 (ifftFlag=1) 变换的标志位

bitReverseFlag: 设置输出位翻转 (bitReverseFlag=1) 或者不翻转 (bitReverseFlag=0) 的标志位

返回值:

无

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)
```

参数:

*S: 指向 CFFT 结构体实例

*p1: 指向复数数据缓存，大小是 2*fftLen. 原地处理

ifftFlag: 设置是正向 (ifftFlag=0) 还是逆向 (ifftFlag=1) 变换的标志位

bitReverseFlag: 设置输出位翻转 (bitReverseFlag=1) 或者不翻转 (bitReverseFlag=0) 的标志位

返回值:

无

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)
```

参数:

*S: 指向 CFFT 结构体实例

*p1: 指向复数数据缓存，大小是 2*fftLen. 原地处理

ifftFlag: 设置是正向 (ifftFlag=0) 还是逆向 (ifftFlag=1) 变换的标志位

bitReverseFlag: 设置输出位翻转 (bitReverseFlag=1) 或者不翻转 (bitReverseFlag=0) 的标志位

返回值:

无

11.2 实数 FFT 函数

11.2.1 函数

- *csi_rfft_fast_f32* : 浮点实数 FFT 处理函数
- *csi_rfft_q15* : Q15 实数 FFT 处理函数
- *csi_rfft_q31* : Q31 实数 FFT 处理函数
- *csi_rfft_fast_init_f32* : 浮点实数 FFT 初始化函数
- *csi_rfft_init_q15* : Q15 RFFT/RIFFT 初始化函数
- *csi_rfft_init_q31* : Q31 RFFT/RIFFT 初始化函数

11.2.2 简要说明

CSI DSP 库为计算实数数据序列的 FFT 实现了特别的算法。FFT 的定义包括了复数数据，但是很多应用的输入只是实数。实数 FFT 算法有对称性的优点，相同长度下比复数算法有速度优势。

快速 RFFT 算法继承自混合基的 CFFT，但是减少了计算量。

长度为 N 的正向实数 FFT 序列计算使用的步骤如下:

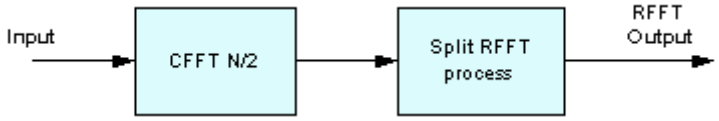


图 11.1: 实数快速傅里叶变换

实数序列初始化的时候，跟 CFFT 的复数处理一样. 之后，一个处理步骤重新整理数据，获取复数频谱的一半。除了第一个复数值包括了两个实数值 X[0] 和 X[N/2]，其他所有的数据都是复数. 换句话说，第一个复数样本包装了两个实数值。

逆 RFFT 的输入应该保持和正向 RFFT 的结果相同的格式。第一步处理步骤为逆 CFFT 预处理数据。

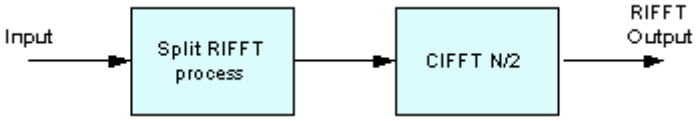


图 11.2: 实数逆向快速傅里叶变换

浮点，Q15 和 Q31 的算法略有不同，我们依次描述各个算法。

浮点

主要的函数是 csi_rfft_fast_f32() 和 csi_rfft_fast_init_f32().

一个 N-点序列 FFT 在频域是偶对称的。数据的第二半等于在频率上被翻转的第一半的共轭:

```
*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]
```

这些数据，我们可以统一的表示 FFT 为

```
*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
```

更近一步，第一个和最后一个样本的虚部，可以用零填充。因此，我们可以将 N 点实数序列表示为 (N/2+1) 复数值:

```
*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.
```

实数 FFT 函数将频域数据包装成这种方式。正向变换以这种方式输出数据，逆向变换期望接收的输入也是这种方式。函数总是根据需要处理位翻转，所有输入和输出总是正常顺序。函数支持 [32, 64, 128, ⋯,8192] 长度的样本.

正向和逆向实数 FFT 函数应用标准 FFT; 正向变换不需要缩放，逆向变换缩放 1/fftLen 。

Q15 和 Q31

实数算法相似的方式定义实数算法，并且应用了（N/2+1）复数变换。

复数变换内部使用缩放防止定点溢出，总共的缩放等于 1/(fftLen/2).

每个变换分别有单独的结构体实例，但是旋转因子和位翻转表可以复用。

使用方式

RFFT 函数有两种不同的使用方式，推荐方法直接使用对应函数，不需要调用初始化函数。这样做，既可以节省内存的使用，还可以简化使用方式。

另一种方法需要在使用前调用相应的初始化函数。初始化函数处理下列操作:

- 设置内部结构体字段值.
- 初始化选择因子表和位翻转表指针.
- 初始化中间复数 FFT 数据结构.

这两种方法都需要初始化，不同的是，推荐方法使用的结构体实例已经预先手动初始化好了，而方法而需要调用初始化函数进行初始化。虽然这两种方法都可以使用，但是推荐使用第一种。结构体实例就要如下手动初始化：

```
*csi_rfft_instance_q31 S = {fftLenReal, fftLenBy2, ifftFlagR,␣
↪bitReverseFlagR, twidCoefRModifier, pTwiddleAReal, pCfft};
*csi_rfft_instance_q15 S = {fftLenReal, fftLenBy2, ifftFlagR,␣
↪bitReverseFlagR, twidCoefRModifier, pTwiddleAReal, pCfft};
```

其中 fftLenReal 是实数变换的长度; fftLenBy2 是中间复数变换的长度. ifftFlagR 选择正向 (=0) 或逆向 (=1) 变换. bitReverseFlagR 选择输出位翻转 (=0) 或者输出正常顺序 (=1). twidCoefRModifier 旋转因子表的步幅调节。这个值基于 FFT 长度； pTwiddleAReal 指向旋转系数 A 数组； pCfft 指向 CFFT 结构体实例. CFFT 结构体也必须被初始化复数结构体实例初始化详细参考 csi_cfft_radix4_f32()

11.2.3 函数说明

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)
```

参数:

`*S`: 指向一个 RFFT 结构体

`*p`: 指向输入数据

`*pOut`: 指向输出数据

`ifftFlag`: RFFT 如果标志位是 0, RIFFT 如果标志位是 1

返回值:

无

推荐使用方式:

针对不同的长度，用法如下:

```
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)
```

参数:

*S: 指向一个 RFFT 结构体

*p: 指向输入数据

*pOut: 指向输出数据

返回值:

无

推荐使用方式:

针对不同的长度，用法如下:

```
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);
```

输入和输出格式:

每个步骤的内部输入缩小 2，以防止 CFFT/CIFFT 处理中饱和。因此，不同的 RFFT 大小的输出格式不同。不同大小的 RFFT 输入和输出格式，下表罗列了 RFFT 和 RIFFT 的缩放位数:

RFFT 大小	输入格式	输出格式	向上缩放的位数
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 大小	输入格式	输出格式	向上缩放的位数
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)
```

参数:

`*S`: 指向一个 RFFT 结构体

`*p`: 指向输入数据

`*pOut`: 指向输出数据

返回值:

无

推荐使用方式:

针对不同的长度，用法如下:

```
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);
```

输入和输出格式:

每个步骤的内部输入缩小 2，以防止 CFFT/CIFFT 处理中饱和。因此，不同的 RFFT 大小的输出格式不同。不同大小的 RFFT 输入和输出格式，下表罗列了 RFFT 和 RIFFT 的缩放位数:

RFFT 大小	输入格式	输出格式	向上缩放的位数
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 大小	输入格式	输出格式	向上缩放的位数
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_fast_init_f32

```
csi_status csi_rfft_fast_init_f32 (csi_rfft_fast_instance_f32 *S, uint16_t fftLen)
```

参数:

`*S`: 指向一个 RFFT 结构体

`fftLen`: 实数序列的长度

返回值:

初始化成功，则返回 CSKY_MATH_SUCCESS ，如果 fftLen 是一个不支持的值，则返回 CSKY_MATH_ARGUMENT_ERROR

简要说明:

参数 fftLen 执行 RFFT/CIFFT 处理的长度. FFT 支持的长度是 32, 64, 128, 256, 512, 1024, 2048, 4096.

这个函数也初始化旋转因子表的指针和位翻转表的指针。

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)
```

参数:

*S: 指向一个 RFFT 结构体

fftLenReal: FFT 的长度

ifftFlagR: 标志位选择正向 (ifftFlagR=0) 或者逆向 (ifftFlagR=1) 变换

bitReverseFlag: 标识位选择输出位翻转 (bitReverseFlag=1) 或者不翻转 (bitReverseFlag=0)

返回值:

初始化成功，则函数返回 CSKY_MATH_SUCCESS ，如果 fftLenReal 不是支持的长度，则返回 CSKY_MATH_ARGUMENT_ERROR。

简要说明:

参数 fftLenReal 指定 RFFT/RIFFT 处理的长度. 支持的 FFT 长度有 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192.

参数 ifftFlagR 控制正向或者逆向计算. 设置 (=1) ifftFlagR 计算 RIFFT, 否则计算 RFFT.

参数 bitReverseFlag 控制输出是正常顺序或者位翻转顺序。设置 (=1) bitReverseFlag ，输出是正常顺序，否则，输出是说字节翻转顺序。

函数也初始化旋转因子表。

注意

初始化函数推荐不要使用，可以使用手动初始化的结构体实例来直接调用函数 csi_rfft_q15()。

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)
```

参数:

*S: 指向一个 RFFT 结构体

fftLenReal: FFT 的长度

ifftFlagR: 标志位选择正向 (ifftFlagR=0) 或者逆向 (ifftFlagR=1) 变换

bitReverseFlag: 标识位选择输出位翻转 (bitReverseFlag=1) 或者不翻转 (bitReverseFlag=0)

返回值:

初始化成功，则返回 CSKY_MATH_SUCCESS，如果 fftLenReal 的长度是不支持的值，则返回 CSKY_MATH_ARGUMENT_ERROR

简要说明:

参数 fftLenReal 执行 RFFT/RIFFT 处理的长度. 支持的 FFT 长度有 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192.

参数 ifftFlagR 控制计算正向或者逆向变换。设置 (=1) ifftFlagR 则计算 RIFFT, 否则计算 RFFT .

参数 bitReverseFlag 控制输出是正常顺序或者位翻转顺序。设置 (=1) bitReverseFlag 则输出是正常顺序，否则输出是位翻转顺序。

函数也初始化旋转因子表。

注意

初始化函数推荐不要使用，可以使用手动初始化的结构体实例来直接调用函数 csi_rfft_q31()。

11.3 DCT IV 型函数

11.3.1 函数

- *csi_dct4_f32* : 浮点 DCT4/IDCT4 处理函数
- *csi_dct4_q15* : Q15 DCT4/IDCT4 处理函数
- *csi_dct4_q31* : Q31 DCT4/IDCT4 处理函数
- *csi_dct4_init_f32* : 浮点 DCT4/IDCT4 初始化函数
- *csi_dct4_init_q15* : Q15 DCT4/IDCT4 初始化函数
- *csi_dct4_init_q31* : Q31 DCT4/IDCT4 初始化函数

11.3.2 简要说明

离散余弦变换 (DCT) 可以用来构建出能量集中在光谱较低部分的输出。意味着可以用最小的数值来表示信号，在存储和传输中都很有用。因此广泛应用在信号和图片编码程序中。DCT 系列 (DCT 类型- 1,2,3,4) 是均匀边界条件的不同组合结果。DCT 系列有出色的能量包装特性，特别是在数据压缩方面。

DCT 本质上是一个实数信号偶扩展的离散傅里叶变换 (DFT)。输入数据的重新排序可以让计算 DCT 变成计算一个实数信号的 DFT，以及少量的附加操作。因此特定硬件和软件就可以按照 DFT 实现一个简单有效的 DCT 算法。

DCT2 型的内部可以用快速傅里叶变换（FFT）实现，由于变换应用在实数数值，因此可以使用实数 FFT 算法。DCT4 的实现则使用了 DCT2，因为他们的实现很相似，只差了一些预处理和后期处理。DCT2 的实现可以描述为以下步骤：

- 输入重排序
- 计算实数 FFT
- 权重乘法和实数 FFT 输出，并从结果中获得实部

综上 DCT4 处理可以用以下框图描述：

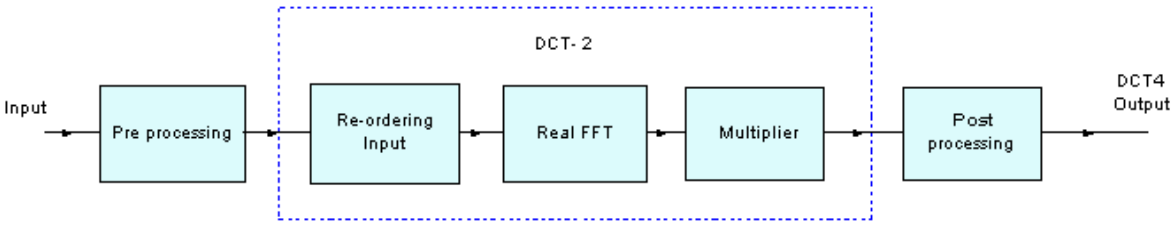


图 11.3: 离散余弦变换 - IV 型

算法:

N-点 IV 型 DCT 公式定义为实数线性转换:

$$X_c(k) = \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} x(n) \cos\left[\left(n + \frac{1}{2}\right)\left(k + \frac{1}{2}\right)\frac{\pi}{N}\right]$$

其中 k = 0,1,2,⋯..N-1

它的逆运算定义为:

$$x(n) = \sqrt{\frac{2}{N}} \sum_{k=0}^{N-1} X_c(k) \cos\left[\left(n+\frac{1}{2}\right)\left(k+\frac{1}{2}\right)\frac{\pi}{N}\right]$$

其中 n = 0,1,2,⋯..N-1

DCT4 矩阵通过乘以 sqrt（2 / N）的总比例因子而变得对合（即它们是自逆的）。变换矩阵的对称性说明用于正向和反向变换计算的快速算法是相同的。注意，DCT4 和逆 DCT4 的实现是相同的，因此，它们可以使用相同的处理函数。

转换支持的长度:

由于 DCT4 内部使用实数 FFT, 它支持所有 csi_rfft_fast_f32() 支持的长度. 为 Q15，Q31 和浮点分别提供了不同的函数。

结构体实例:

实数 FFT 和 FFT 的实例，余弦值表和旋转因子表都保存在一个数据结构的实例中。每个转换都必须定义一个单独的数据结构体实例。为支持的 3 种数据类型分别提供了不同的结构体实例。

使用方式:

和 RFFT 函数类似，DCT4 函数也有两种不同的使用方式，推荐方法直接使用对应函数，不需要调用初始化函数。这样做，既可以节省内存的使用，还可以简化使用方式。

另一种方法需要在使用前调用相应的初始化函数。初始化函数处理下列操作:

- 设置内部结构体字段的值
- 初始化实数 FFT 作为 DCT4 内部使用的处理函数，调用的是 csi_rfft_fast_init_f32().

这两种方法都需要初始化，不同的是，推荐方法使用的结构体实例已经预先手动初始化好了，而方法而需要调用初始化函数进行初始化。虽然这两种方法都可以使用，但是推荐使用第一种。手动初始化结构体实例如下:

```
*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};
```

其中 N 是 DCT4 的长度; Nby2 是 DCT4 长度的一半; normalize 是使用的归一化因子，并且相等于 sqrt(2/N); pTwiddle 指向旋转因子表; pCosFactor 指向余弦因子表; pRfft 指向实数 FFT 实例; pCfft 指向复数 FFT 实例; CFFT 和 RFFT 结构体也需要被初始化，详细情况参考负责初始化的 csi_rfft_fast_f32() .

定点行为:

使用定点 DCT4 转换函数需要注意。特别是要考虑，在每个函数内使用的累加器的溢出和饱和行为。具体参考每个函数各自的文档和使用说明。

11.3.3 函数说明

11.3.3.1 csi_dct4_f32

```
void csi_dct4_f32 (const csi_dct4_instance_f32 *S, float32_t *pState, float32_t␣
↪*pInlineBuffer)
```

参数:

*S: 指向 DCT/IDCT4 结构体实例

*pState: 指向状态数组

*pInlineBuffer: 指向输入和输出数据

返回值:

无

推荐使用方式:

针对不同的长度，用法如下:

```
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)
```

参数:

*S: 指向 DCT/IDCT4 结构体实例

*pState: 指向状态数组

*pInlineBuffer: 指向输入和输出数据

返回值:

无

推荐使用方式:

针对不同的长度，用法如下:

```
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);
```

输入和输出格式:

输入样本需要缩小 1 个位来防止在 Q15 DCT 处理中饱和。RFFT 处理函数的内部输入会被缩放以防止溢出。缩放的位数，依赖于变换的大小。不同 DCT 大小的输入和输出的格式和位的数量，缩放位的数量在下表中:

DCT 大小	输入格式	输出格式	向上缩放的位数
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)
```

参数:

`*S`: 指向 DCT/IDCT4 结构体实例

`*pState`: 指向状态数组

`*pInlineBuffer`: 指向输入和输出数据

返回值:

无

推荐使用方式:

针对不同的长度，用法如下:

```
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);
```

输入和输出格式:

输入样本需要缩小 1 个位来防止在 Q31 DCT 处理中饱和。因为从 DCT2 转换到 DCT4 涉及到一个减法。RFFT 处理函数的内部输入会被缩放以防止溢出。缩放的位数，依赖于变换的大小。不同 DCT 大小的输入输出格式和位的数量，缩放位的数量在下表中:

DCT 大小	输入格式	输出格式	向上缩放的位数
8192	2.30	14.18	13
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)
```

参数:

`*S`: 指向 DCT4/IDCT4 结构体实例

`*S_RFFT`: 指向 RFFT/RIFFT 结构体实例

`*S_CFFT`: 指向 CFFT/CIFFT 结构体实例

`N`: DCT4 的长度

`Nby2`: DCT4 的长度的一半

`normalize`: 归一化因子

返回值:

csi_status 如果初始化成功，则函数返回 CSKY_MATH_SUCCESS ，如果 fftLenReal 是不支持的变换长度，则返回 CSKY_MATH_ARGUMENT_ERROR

归一化因子:

归一化因子是 sqrt(2/N), 依赖于变换大小 N. 下表提供不同的 DCT 大小的浮点归一化因子:

DCT 大小	归一化因子的值
8192	0.015625
2048	0.03125
512	0.0625
128	0.125

注意

初始化函数推荐不要使用，可以使用手动初始化的结构体实例来直接调用函数 csi_dct4_f32()。

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)
```

参数:

*S: 指向 DCT4/IDCT4 结构体实例

*S_RFFT: 指向 RFFT/RIFFT 结构体实例

*S_CFFT: 指向 CFFT/CIFFT 结构体实例

N: DCT4 的长度

Nby2: DCT4 的长度的一半

normalize: 归一化因子

返回值:

csi_status 初始化成功，则函数返回 CSKY_MATH_SUCCESS，如果 N 不是支持的变换长度，则返回 CSKY_MATH_ARGUMENT_ERROR

归一化因子:

归一化因子是 sqrt(2/N), 依赖于变换的大小 N. 归一化因子是 1.15 格式，下表罗列了在不同 DCT 大小的值:

DCT 大小	归一化因子的值 (16 进制)
8192	0x200
2048	0x400
512	0x800
128	0x1000

注意

初始化函数推荐不要使用，可以使用手动初始化的结构体实例来直接调用函数 csi_dct4_q15().

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)
```

参数:

*S: 指向 DCT4/IDCT4 结构体实例

*S_RFFT: 指向 RFFT/RIFFT 结构体实例

*S_CFFT: 指向 CFFT/CIFFT 结构体实例

N: DCT4 的长度

Nby2: DCT4 的长度的一半

normalize: 归一化因子

返回值:

csi_status 初始化成功，则函数返回 CSKY_MATH_SUCCESS，如果 N 不是支持的变换长度，则返回 CSKY_MATH_ARGUMENT_ERROR

归一化因子:

归一化因子是 sqrt(2/N), 依赖于变换的长度 N. 归一化因子是 1.31 格式，下表中罗列了不同 DCT 大小的值:

DCT Size	归一化因子的值 (16 进制)
8192	0x2000000
2048	0x4000000
512	0x8000000
128	0x10000000

注意

初始化函数推荐不要使用，可以使用手动初始化的结构体实例来直接调用函数 csi_dct4_q31()。