User's Manual

32

RX DSP Library APIs Version 5.0

User's Manual: Software

RENESAS MCU
RX Family

Renesas Electronics
www.renesas.com

Rev.1.00 Jan. 2019

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   “Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   “High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.
   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Handling of Unused Pins

   Handle unused pins in accordance with the directions given under Handling of Unused Pins in the manual.

   — The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on

   The state of the product is undefined at the moment when power is supplied.

   — The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the moment when power is supplied.
   In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the moment when power is supplied until the reset process is completed.
   In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the moment when power is supplied until the power reaches the level at which resetting has been specified.

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

   — The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals

   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.

   — When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.

5. Differences between Products

   Before changing from one product to another, i.e. to a product with a different part number, confirm that the change will not lead to problems.

   — The characteristics of Microprocessing unit or Microcontroller unit products in the same group but having a different part number may differ in terms of the internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

How to Use This Manual

1. Objective and Target Users

This manual was written to explain the specifications and usage of this DSP Library to target users, i.e. those who will be using this DSP Library to develop application programs for the RX Family microcontroller. Target users are expected to understand the fundamentals of the C language and the RX Family microcontroller.

This manual is organized in the following items: declaration of API, descriptions of each API, internal functions, and required resource and number of execution cycles of each function.

> When using this DSP Library, take all points to note into account.
> Points to note are given in their contexts and at the final part of each section, and in the section giving usage notes.

> The list of revisions is a summary of major points of revision or addition for earlier versions. It does not cover all revised items. For details on the revised points, see the actual locations in the manual.

2. Notation in this Manual

<intype> and <outttype> are used to indicate the input and output data types this API supports. <intype> is the input data type, and <outttype> is the output data type. The types that are applicable to <intype> and <outttype> are described below.

- i16: 16-bit fixed-point real number
- ci16: 16-bit fixed-point complex number
- i32: 32-bit fixed-point real number
- ci32: 32-bit fixed-point complex number
- f32: Single precision floating-point real number
- cf32: Single precision floating-point complex number

3. List of Abbreviations and Acronyms

Abbreviation	Acronym
API	Application programming interface
DFT	Discrete Fourier transform
DSP	Digital signal processor
FFT	Fast Fourier transform
FIR	Finite impulse response
IDFT	Inverse discrete Fourier transform
IFFT	Inverse fast Fourier transform
IIR	Infinite impulse response
LMS	Least mean square
LSB	Least significant bit
MAD	Mean absolute deviation
MSB	Most significant bit

Table of Contents

1. Overview

This manual contains the specifications for the Renesas RX family DSP Library APIs version 5.0. It describes the overall specifications and guidelines for the DSP Library APIs, as well as detailed specifications related to the DSP algorithms.

1.1 DSP Library Structure

The DSP Library provides eight types of files based on FPU support, endianness, and the presence or absence of error checking. The library filenames corresponding to the respective conditions are shown in Table 1.1.

Table 1.1 DSP Library File Name List

FPU	Endianness	Error Checking	Library File Name
Not supported	Little-endian	None	RX_DSP_NOFPU_LE.lib
		Available	RX_DSP_NOFPU_LE_Check.lib
	Big-endian	None	RX_DSP_NOFPU_BE.lib
		Available	RX_DSP_NOFPU_BE_Check.lib
Supported	Little-endian	None	RX_DSP_FPU_LE.lib
		Available	RX_DSP_FPU_LE_Check.lib
	Big-endian	None	RX_DSP_FPU_BE.lib
		Available	RX_DSP_FPU_BE_Check.lib

1.2 DSP Library APIs

The APIs defined in this document are shown below.

1.2.1 Statistical Operation API

- Mean value
- Maximum and minimum values
- Maximum and minimum values with index
- Maximum and minimum absolute values
- Sample variance
- Histogram
- Mean absolute deviation
- Median

1.2.2 Filter Operation API

- Generic FIR filter
- IIR Biquad filter
- Single-Pole IIR filter

1.2.3 Linear Transform API

- Complex DFT
- Complex IDFT
- Real DFT
- Complex conjugate symmetric IDFT
- Complex FFT

- Complex IFFT
- Real FFT
- Complex conjugate symmetric IFFT

1.2.4 Complex Number Operation API

- Complex magnitude value
- Complex number magnitude squared value
- Complex phase
- Complex addition
- Complex subtraction
- Complex multiplication
- Complex conjugate

1.2.5 Matrix Operation API

- Matrix addition
- Matrix subtraction
- Matrix multiplication
- Matrix transposition
- Matrix real number multiplication

1.3 Precautions on Use

1.3.1 Interrupts During Accumulator Use

Many of the DSP Library APIs use an accumulator (ACC) for operations. In interrupt functions that perform multiplication using the accumulator, save and restore the accumulator to prevent corruption of the results of DSP Library API operations.

Saving of the accumulator is specified using the compiler option save_acc or the #pragma extension (#pragma interrupt acc).

For details, see RX Family C/C++ Compiler Assembler, Optimizing Linkage Editor User's Manual.

1.3.2 Standard Library

When using an API that performs floating-point operations, it is necessary to link to the standard library below.

API Performing Floating-Point Operations	Required Standard Library
Complex number kernel (complex)	mathf.h
Conversion kernel (transform)	

1.3.3 Floating-Point Exceptions

During floating-point operations, the DSP Library does not perform floating-point error checking on coefficients and input data. The RX Family includes an FPU that supports IEEE 754 exceptions. It is recommended that users use a floating-point exception that handles FPU errors. For information regarding floating-point exception settings, see the respective RX Series user's manuals.

1.3.4 Floating-Point NaN (Not a Number) Checking

The DSP Library does not perform floating-point NaN checking. Be sure that all floating-point values are valid IEEE 754

single precision floating-point values.

2. DSP Library APIs

This section describes the specifications that are common to all of the APIs of the DSP Library.

2.1 Terms

The following terms are used in this manual.

- Function: the respective functions in the APIs of the DSP Library that are called by the user.
- Vector: a one-dimensional array of data.
- Matrix: a two-dimensional array of data.
- Element or entry: a single item in a vector or matrix.

If two vectors have the same number of elements, they are said to have the same length. If two matrices have the same number of rows and columns, they are said to have the same dimensions.

If multiple vectors have the same length, the elements of one of the vectors correspond to the respective elements of the others. If multiple matrices have the same dimensions, the elements of one of the matrices correspond to the respective elements of the others.

2.2 Data Types

The DSP Library APIs handle the following three data types.

- 16-bit fixed-point
- 32-bit fixed-point
- 32-bit floating-point

The DSP Library APIs support real numbers and complex numbers using the above three data types.

The format of fixed-point is shown as Qx.yy. "x" shows the number of bits of sign and integral part. "yy" shows the number of bits after decimal point. For example, "Q1.15" shows 16-bit fixed-point and a range of values is [-1.0,1.0].

In addition, there are functions that produce 32-bit fixed-point output from 16-bit fixed-point input.

The input data, output data, and coefficients in all APIs in the DSP Library are always signed values, even if it is known that only positive values are being handled. For example, the mean absolute value API only outputs positive values, but is always used with signed data.

For API parameters, use unsigned values. For example, define the number of taps in the FIR filter using an unsigned value.

2.3 Structures

The DSP Library APIs define the following types of structures.

- Complex numbers
- Vectors and matrices
- Handles

2.3.1 Complex Numbers

Complex number data is composed of two data values as a single complex data element. The following three types are defined.

- 16-bit fixed-point complex number structure: cplxi16_t

- 32-bit fixed-point complex number structure: cplxi32_t
- 32-bit floating-point complex number structure: cplxf32_t

The definition of the 16-bit fixed-point complex number structure is shown below. The other complex number structures are defined in the same way.

```
typedef struct
{
   int16_t re;  /* real part of complex */
   int16_t im;  /* imaginary part of complex */
} cplxi16_t;
```

2.3.2 Vectors and Matrices

The vector structure is defined as follows, using a vector element count and a pointer to a data array.

```
typedef struct
{
   uint32_t n;      /* number of elements */
   void *   data;   /* pointer to the first element of data */
} vector_t;
```

The matrix structure is defined as follows, using a row count, a column count, and a pointer to a data array.

```
typedef struct
{
   uint16_t nRows;     /* number of rows */
   uint16_t nCols;     /* number of colums */
   void *   data;      /* pointer to the first element of data */
} matrix_t;
```

The user must allocate the memory for the vector or matrix data. In addition, the “data” element of the vector and matrix structures is declared as (void*), so it is not necessary to separate the vector and matrix structures based on the data types supported by the libraries. The above structures can be used in any combination of real and complex number types and fixed-point and floating-point types.

For example, a N elements, 16-bit fix-point complex number vector is created and initialized as follows.

```
#define   N  (4)
vector_t  my_cplxi16_vector; // instantiate the vector
cplxi16_t vector_data[N];
int16_t j;

my_cplxi16_vector.n = N;
my_cplxi16_vector.data = vector_data;
/* Assuming that we’ll be using Q1.15 fractional format, we’re initializing */
/* the vector to the complex value 0.5 + 0i */
for (j = 0; j < N; j++)
{
   vector_data[j].re = (1 << 14);
   vector_data[j].im = 0;
}
```

2.3.3 Handles

The memory, constants, and runtime parameters required by functions are defined using handle data structures specified by the respective APIs. For example, the FIR filter handle is defined as follows.

```
typdef struct
{
   uint32_t taps;      /* number of filter taps */
   void *coefs;        /* pointer to filter coefficients */
   void *state;        /* pointer to filter state data, including the filter's
                       delay line and any other implementation-dependent state */
   scale_t scale;      /* scaling parameter */
   uint32_t options;   /* option flags that specify rounding, saturation,
                          or other behaviors */
} r_dsp_firfilter_t;
```

All members of handle data structures must be initialized by the user. This includes coefficients and pointers to memory as well. Coefficients and memory are provided by the user. There are some functions return the required memory size. Depending on the API, use them to allocate the memory and set the handle members.

In addition, many APIs set runtime parameters according to the "options" member of the handle data structure.

The parameters provided by the handle data structure cannot be changed by the user without performing initialization using the initialization function of the API. For example, changing the rounding mode and tap count are prohibited during a FIR filter operation. In order to change these parameters, the user must allocate a sufficient amount of memory for the internal state of the API and for the provided parameters, and perform initialization again using the initialization function of the API.

2.3.4 Scaling Parameter Union

The following union is used as the scaling parameter for the filter operation API and the matrix operation API. The member of union to set the value depends on the data types of the respective APIs.

```
typedef union
{
   float   f32;
   int32_t i32;
} scale_t;
```

For the scaling, refer to 2.9 Scaling.

2.4 Function Name Rules

All functions are named according to the following format.

```
R_DSP_<API>[_variant][_function]_<intype><outtype>(arguments)
```

or

```
R_DSP_<API>[_variant][_function]_<intype>(arguments)
```

The individual components of the above format are as follows.

- "R_DSP_" is the prefix of DSP Library function names.
- <API> is the name of the API. Example: For the FIR filter, it is "FIR".
- [_variant] is a variant of a function. Example: The complex number magnitude API has a high-speed variant called "_Fast".
- [_Function] is a function that is other than calculation operation such as configuration and management. Example: The FIR filter has a function called "_Init" that initializes delay memory.

- <intype> and <outtype> indicate the respective data types of the input and output. The specifiers are as follows.
- i16: 16-bit fixed-point data
- ci16: 16-bit fixed-point complex number data
- i32: 32-bit fixed-point data
- ci32: 32-bit fixed-point complex number data
- f32: 32-bit floating-point data
- cf32: 32-bit floating-point complex number data

In the DSP Library, the input and output data types are specified by the respective API functions.

However, for functions for which it is self-evident that the input and output data types are the same, only <intype> is used in the function name, and <outtype> is omitted. For example, for functions that get the maximum and minimum values in a vector, the result will be the same data type as the elements in the input vector, so only <intype> will be used in the function name.

2.5 Arguments and Return Values

Functions define arguments in the following order.

Functions for which the return value is the function execution result (e.g., an error), a memory size, or the like have the following format.

```
<status/size> function(<handle>, <input1>,…, <inputN>, <output1>,…,<outputN>,
<additional options>)
```

Functions for which the return value is the operation result have the following format.

```
<type> function(<input1>,…, <inputN>, <additional options>)
```

Here,

- <status/size>: “status” is the r_dsp_status_t type and “size” is the number of bytes
- <type>: the type of the return value
- <handle>: a pointer to the handle data structure specified by the function
- <input1> ... <inputN>: one or more scalar values or pointers
- <output1> ... <outputN>: one or more output pointers
- <additional options>: optional parameters that are not contained in the handle data structure

Functions do not necessarily have all of the above arguments. For example, the matrix addition function does not have memory and coefficients, so it does not have a handle data structure. There are also functions like the FIR filter initialization function with neither input nor output.

Most functions return a return value that is type r_dsp_status_t, which is a 32-bit integer. The return value will be, for example, an error code, or the function’s operation result. For example, the R_DSP_IIRBiquad_StateSize function in the IIR Biquad filter API returns the amount of memory required for delay memory, and returns an error if the parameter for memory size calculation is not sufficient.

In addition, a function which calculates a single real number value returns that operation result.

2.6 Header Files

The header files in the DSP Library are divided according to API category. The followings are the categories and header files.

- Statistical Function API: r_dsp_statistical.h
- Filter Function: r_dsp_filters.h
- Linear Transform Function API: r_dsp_transform.h

- Complex Function API: r_dsp_complex.h
- Matrix Function API: r_dsp_matrix.h

The functions in the respective categories depend upon the data structures and data types that are common to the DSP Library APIs, so the functions reference the shared header file r_dsp_types.h. The shared definitions in the DSP Library are all in r_dsp_types.h.

2.7 Error Handling

In the library with error checking, all functions check arguments and parameters as much as possible. With the exception of functions that return the memory size required by the API, most functions have return values of type r_dsp_status_t, and return the execution result as a code. The codes are defined as follows.

- R_DSP_STATUS_<description>: a status code. 0 or positive. The operation executed to completion.
- R_DSP_ERR_<description>: an error code. Negative. The operation was aborted.

An error code indicates the conditions under which the operation of the function was terminated, for example the detection of a null pointer input. In contrast, a status code indicates proceeded operation, but has a significant effect on the operation result. For example, an arithmetic overflow is indicated with a status code. As a result, the user can programmatically distinguish between two types of conditions corresponding to the sign of the code.

Other functions return an operation result or an error code. For example, the R_DSP_IIRBiquad_StateSize_<intype><outtype> function returns the memory size required as delay memory by the IIR filter, but will return an error code if the memory size cannot be calculated.

The error codes and status codes are defined by an enum declaration in the r_dsp_types.h header file. These are listed in Table 2.1.

Table 2.1 Error Codes and Status Codes

Error/Status Definition	Value	Description
R_DSP_STATUS_OK	0	Normal exit
R_DSP_STATUS_HISTO_OUT_OF_RANGE	3	Floor value ((x - offset) * scale) is negative or greater than or equal to histo.n
R_DSP_STATUS_UNDEFINED_RESULT	2	Size 0 vector element detected
R_DSP_STATUS_OVERFLOW	1	Numeric overflow occurred
R_DSP_ERR_HANDLE_NULL	-100	Handle pointer is null
R_DSP_ERR_INPUT_NULL	-99	Input vector/matrix, data, or data size pointer is null
R_DSP_ERR_OUTPUT_NULL	-98	Output vector/matrix or data pointer is null
R_DSP_ERR_STATE_NULL	-97	Memory pointer is null
R_DSP_ERR_COEFF_NULL	-96	Coefficient vector pointer is null
R_DSP_ERR_REFER_NULL	-95	Reference sample pointer is null (Leaky LMS filter)
R_DSP_ERR_INVALID_INPUT_SIZE	-94	Input vector/matrix dimensions are outside specifiable range
R_DSP_ERR_INVALID_OUTPUT_SIZE	-93	Output vector/matrix dimensions are insufficient
R_DSP_ERR_INVALID_TAPS	-92	Number of filter taps is outside specifiable range
R_DSP_ERR_INVALID_STAGES	-91	Number of filters is outside specifiable range
R_DSP_ERR_INVALID_OPTIONS	-90	Mode not supported by handle options specified
R_DSP_ERR_INVALID_SCALE	-89	Scaling parameter is outside specifiable range
R_DSP_ERR_DIMENSIONS	-88	Matrix dimensions are insufficient or mismatched
R_DSP_ERR_INVALID_POINTS	-87	Number of conversion points is outside specifiable range
R_DSP_ERR_NO_MEMORY_AVAILABLE	-86	Required memory is not allocated
R_DSP_ERR_INVALID_QINT	-85	Number of specified integer bits is outside specifiable range
R_DSP_ERR_INVALID_COEFF	-84	Coefficient is outside specifiable range

2.8 Rounding and Saturation

The fixed-point input/output functions in the DSP Library support a rounding mode and a saturation mode. Rounding mode and saturation mode are specified in the “options” member of the handle structures of the respective APIs. The specifiers to be specified in “options” for rounding mode and saturation mode are shown in Table 2.2.

Table 2.2 Rounding and Saturation Specifiers

Bit Field In Option	Mode	Specifier	Value	Description
2-0	Rounding	R_DSP_ROUNDING_DEFAULT	0	Prescribed by each API
		R_DSP_ROUNDING_TRUNC	1	Truncate
		R_DSP_ROUNDING_NEAREST	2	Round to nearest value
4-3	Saturation	R_DSP_SATURATION_DEFAULT	0	Prescribed by each API
		R_DSP_NOSATURATE	1	Do not perform clipping
		R_DSP_SATURATE	2	Perform clipping

R_DSP_ROUNDING_DEFAULT and R_DSP_SATURATION_DEFAULT will result in the default processing of each individual API. For example, the leaky LMS filter will fail to converge without rounding, so its default is to use rounding. In contrast, the FIR filter does not require rounding, and defaults to performing truncation.

2.9 Scaling

The filter function API and matrix function API provide scaling capability. With scaling, a constant parameter is applied as multiply factor to the operation results of the respective APIs.

The scaling parameter is specified using the scaling parameter union of the handle data structure of each API.

2.9.1 Fixed-Point Scaling

With fixed-point scaling, multiplication is performed. For API operation results of Qx.yy format, a right-shift amount is specified such that the number of bits (yy) after the decimal point becomes the desired number of bits.

For example, for the 32-bit FIR filter, if the input data is Q1.31 and the coefficient is Q1.31, the internal operation result will be a 64-bit Q2.62. To obtain a Q1.31 result, a right shift of 62 bits – 31 bits = 31 bits is necessary because 32-bit FIR filter outputs the result in LSB aligned. Thus, in this case, the scaling coefficient will be set to 31.

2.9.2 Floating-Point Scaling

With floating-point scaling, the value with which to multiply the API operation results is specified as a floating-point value. The scaling coefficient is specified as a positive number.

3. Common API

This section describes the following shared function, which is used in the DSP Library.

Version Acquisition API

3.1 Version Acquisition API

This function obtains version and variation information for the RX DSP Library.

Format

```
uint32_t R_DSP_GetVersion(void)
```

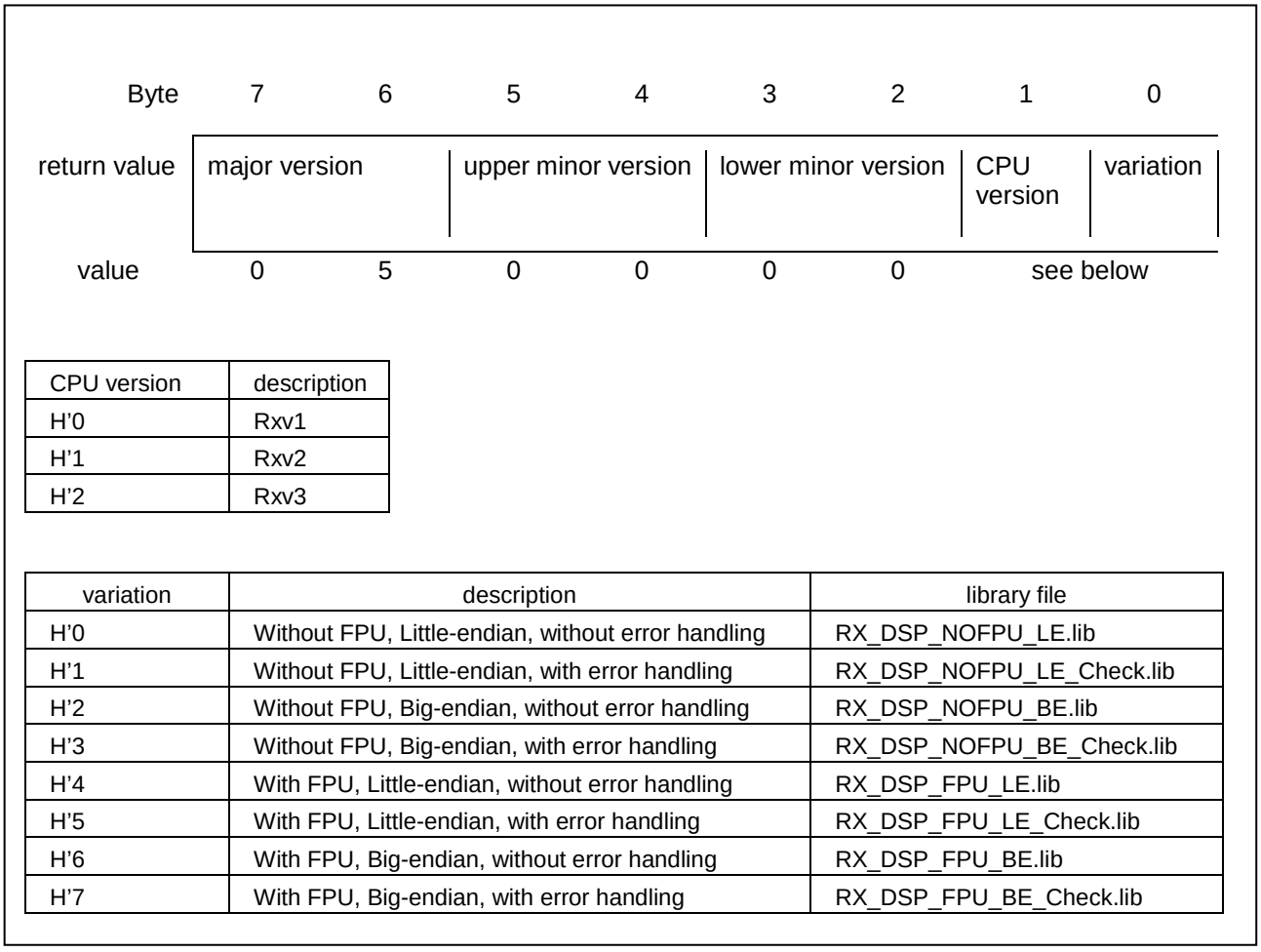
Arguments

None

Return Values

Returns the version number and variation information as a 32-bit integer. Figure 3.1 shows the structure of the return value.

Figure 3.1 Structure of Version Number and Variation Information



CPU version	description
H'0	Rxv1
H'1	Rxv2
H'2	Rxv3

variation	description	library file
H'0	Without FPU, Little-endian, without error handling	RX_DSP_NOFPU_LE.lib
H'1	Without FPU, Little-endian, with error handling	RX_DSP_NOFPU_LE_Check.lib
H'2	Without FPU, Big-endian, without error handling	RX_DSP_NOFPU_BE.lib
H'3	Without FPU, Big-endian, with error handling	RX_DSP_NOFPU_BE_Check.lib
H'4	With FPU, Little-endian, without error handling	RX_DSP_FPU_LE.lib
H'5	With FPU, Little-endian, with error handling	RX_DSP_FPU_LE_Check.lib
H'6	With FPU, Big-endian, without error handling	RX_DSP_FPU_BE.lib
H'7	With FPU, Big-endian, with error handling	RX_DSP_FPU_BE_Check.lib

Example

```
#include <r_dsp_types.h>
#define MIN_VERSION       0x05000000    // Version 5.0.0
uint32_t        my_version;

/* Get Library Version & Variant information */
my_version = R_DSP_GetVersion();
if ((MIN_VERSION > (my_version & R_DSP_VERSION_INFO_MASK)) ||\
   (R_DSP_CPU_V2 != (my_version & R_DSP_CPU_INFO_MASK)) ||\
   (NOFPU_LE_NOCHECK != (my_version & R_DSP_VARIATION_INFO_MASK)))  // correct version &
variant?
 {
runtime_err_function(WRONG_DSPLIB_VERSION); // user defined err function
}
```

4. Statistics Operation API

This section describes the statistics functions, which are supported by the DSP Library.

- Mean value functions
- Minimum value functions
- Maximum value functions
- Minimum value with index functions
- Maximum value with index functions
- Mean absolute value and maximum absolute value functions
- Mean value and sample variance functions
- Sample variance functions
- Histogram functions
- Mean value and mean absolute deviation (MAD) functions
- Mean absolute deviation (MAD) functions
- Median functions (working memory not required)
- Median functions (working memory used)

4.1 Mean Value Functions

These functions calculate the mean value of a real number array.

Format and Function List

```
r_dsp_status_t R_DSP_Mean_<intype>(
   const vector_t * input,
   <outtype> * mean
)
```

<intype> can be i16, i32, or f32.

<outtype> must be an int16_t, int32_t, or float of the same type as <intype>.

Table 4.1 shows the functions for the respective combinations of <intype> and <outtype>.

Table 4.1 Mean Value Function List

<intype>	<outtype>	Function
i16	int16_t	R_DSP_Mean_i16(const vector_t * input, int16_t * mean)
i32	int32_t	R_DSP_Mean_i32(const vector_t * input, int32_t * mean)
f32	float	R_DSP_Mean_f32(const vector_t * input, float * mean)

Arguments

input	Pointer to a vector_t that stores the input data. The referenced members are as follows.
input->n	Input data count. It supports up to 65536.
input->data	Pointer to the first element of a real number array.
mean	Pointer to a variable that stores the operation result of these functions.

Return Value

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Pointer to input vector is null, or data member of the vector is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is outside specifiable range.
R_DSP_ERR_OUTPUT_NULL	Pointer to output is null.

Description

These functions calculate the mean value of an input real number array. The mean value is calculated according to the formula below.

$$\frac{1}{N}\sum_{i=0}^{N-1} x_i$$

Here, x_i represents the *i*th input sample, and N represents the sample count.

Example

```
#include <r_dsp_statistical.h>
float  array[65535];
float mean;
r_dsp_status_t my_sts = R_DSP_STATUS_OK; // place to return status
vector_t v = {65535, array};  // Fill it up with something interesting ...

/* Call the library function */
my_sts = R_DSP_Mean_f32(&v, &mean);
```

4.2 Minimum Value Functions

These functions find the minimum value in a real number array.

Format and Function List

```
r_dsp_status_t R_DSP_Min_<intype>(
   const vector_t * input,
   <outtype> * min
)
```

<intype> can be i16, i32, or f32.

<outtype> must be an int16_t, int32_t, or float of the same type as <intype>.

Table 4.2 shows the functions for the respective combinations of <intype> and <outtype>.

Table 4.2 Minimum Value Function List

<intype>	<outtype>	Function
i16	int16_t	R_DSP_Min_i16(const vector_t * input, int16_t * min)
i32	int32_t	R_DSP_Min_i32(const vector_t * input, int32_t * min)
f32	float	R_DSP_Min_f32(const vector_t * input, float * min)

Arguments

input	Pointer to a vector_t that stores the input data. The referenced members are as follows.
input->n	Input data count. It supports up to 65536.
input->data	Pointer to the first element of a real number array.
min	Pointer to a variable that stores the minimum value found by these functions.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Pointer to input vector or data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is outside specifiable range.
R_DSP_ERR_OUTPUT_NULL	Pointer to output is null.

Example

```
#include <r_dsp_statistical.h>

float  array[65535];
float  min;
r_dsp_status_t   my_sts = R_DSP_STATUS_OK; // place to return status

/* fill it up with something interesting ... */
vector_t v = {65535, array};

/* Call the library function */
my_sts = R_DSP_Min_f32(&v, &min);
```

4.3 Maximum Value Functions

These functions find the maximum value in a real number array.

Format and Function List

```
r_dsp_status_t R_DSP_Max_<intype>(
   const vector_t * input,
   <outtype> * max
)
```

<intype> can be i16, i32, or f32.

<outtype> must be an int16_t, int32_t, or float of the same type as <intype>.

Table 4.3 shows the functions for the respective combinations of <intype> and <outtype>.

Table 4.3 Maximum Value Function List

<intype>	<outtype>	Function
i16	int16_t	R_DSP_Max_i16(const vector_t * input, int16_t * max)
i32	int32_t	R_DSP_Max_i32(const vector_t * input, int32_t * max)
f32	float	R_DSP_Max_f32(const vector_t * input, float * max)

Arguments

input	Pointer to a vector_t that stores the input data. The referenced members are as follows.
input->n	Input data count. It supports up to 65536.
input->data	Pointer to the first element of a real number array.
max	Pointer to a variable that stores the maximum value found by these functions.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Pointer to input vector or data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is outside specifiable range.
R_DSP_ERR_OUTPUT_NULL	Pointer to output is null.

Example

```
#include <r_dsp_statistical.h>

float array[65535];
float max;

r_dsp_status_t   my_sts = R_DSP_STATUS_OK; // place to return status

/* fill it up with something interesting ... */
vector_t v = {65535, array};

/* Call the library function */
my_sts = R_DSP_Max_f32(&v, &max);
```

4.4 Minimum Value with Index Functions

These functions find the minimum value in a real number array, as well as the array index of the minimum value.

Format and Function List

```
r_dsp_status_t R_DSP_ArgMin_<intype>(
   const vector_t * input,
   <outtype> * minval,
   uint16_t * imin
)
```

<intype> can be i16, i32, or f32.

<outtype> must be an int16_t, int32_t, or float of the same type as <intype>.

The data type of the variable that stores the index of the minimum value is uint16_t.

Table 4.4 shows the functions for the respective combinations of <intype> and <outtype>.

Table 4.4 Indexed Minimum Value Function List

<intype>	<outtype>	Function
i16	int16_t	R_DSP_ArgMin_i16(const vector_t * input, int16_t * minval, uint16_t * imin)
i32	int32_t	R_DSP_ArgMin_i32(const vector_t * input, int32_t * minval, uint16_t * imin)
f32	float	R_DSP_ArgMin_f32(const vector_t * input, float * minval, uint16_t * imin)

Arguments

input	Pointer to a vector_t that stores the input data. The referenced members are as follows.
input->n	Input data count. It supports up to 65535.
input->data	Pointer to the first element of a real number array.
minval	Pointer to a variable that stores the minimum value found by these functions.
imin	Pointer to a variable that stores the index in the real number array where the minimum value is located.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Pointer to input vector or data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is outside specifiable range.
R_DSP_ERR_OUTPUT_NULL	Pointer to output is null.

Example

```
#include <r_dsp_statistical.h>

int16_t   x[5] = {5, 2, 7, 1, 99};
int16_t   minval;
uint16_t  imin;
vector_t  vx = {5, x};

r_dsp_status_t my_sts = R_DSP_STATUS_OK; // place to return status

my_sts = R_DSP_ArgMin_i16 (&vx, &minval, &imin);
/* minval = 1 */
/* imin = 3 */
```

4.5 Maximum Value with Index Functions

These functions find the maximum value in a real number array, as well as the array index of the maximum value.

Format and Function List

```
r_dsp_status_t R_DSP_ArgMax_<intype>(
   const vector_t * input,
   <outtype> * maxval,
   uint16_t * imax
)
```

<intype> can be i16, i32, or f32.

<outtype> must be an int16_t, int32_t, or float of the same type as <intype>.

The data type of the variable that stores the index of the maximum value is uint16_t.

Table 4.5 shows the functions for the respective combinations of <intype> and <outtype>.

Table 4.5 Indexed Maximum Value Function List

<intype>	<outtype>	Function
i16	int16_t	R_DSP_ArgMax_i16(const vector_t * input, int16_t * maxval, uint16_t * imax)
i32	int32_t	R_DSP_ArgMax_i32(const vector_t * input, int32_t * maxval, uint16_t * imax)
f32	float	R_DSP_ArgMax_f32(const vector_t * input, float * maxval, uint16_t * imax)

Arguments

input	Pointer to a vector_t that stores the input data. The referenced members are as follows.
input->n	Input data count. It supports up to 65536.
input->data	Pointer to the first element of a real number array.
maxval	Pointer to a variable that stores the maximum value found by these functions.
imax	Pointer to a variable that stores the index in the real number array where the maximum value is located.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Pointer to input vector or data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is outside specifiable range.
R_DSP_ERR_OUTPUT_NULL	Pointer to output is null.

Example

```
#include <r_dsp_statistical.h>

int16_t   x[5] = {5, 2, 7, 1, 99};
int16_t   maxval;
uint16_t  imax;
vector_t  vx = {5, x};

r_dsp_status_t my_sts = R_DSP_STATUS_OK; // place to return status

my_sts = R_DSP_ArgMax_i16 (&vx, &maxval, &imax);
/* maxval = 99 */
/* imax = 4 */
```

4.6 Mean Absolute Value and Maximum Absolute Value Functions

These functions calculate the mean value and maximum value of the absolute values of the real numbers in an input array.

Format and Function List

```
r_dsp_status_t R_DSP_MeanAbs_<intype>(const vector_t * input, <outtype> * mean)
r_dsp_status_t R_DSP_MaxAbs_<intype>(const vector_t * input, <outtype> * max)
r_dsp_status_t R_DSP_MeanMaxAbs_<intype>(const vector_t * input, <outtype> * mean,
<outtype> * max)
```

<intype> can be i16, i32, or f32.

<outtype> must be an int16_t, int32_t, or float of the same type as <intype>.

Table 4.6 shows the functions for the respective combinations of <intype> and <outtype>.

Table 4.6 Mean Absolute Value and Maximum Absolute Value Function List

Operation	<intype>	<outtype>	Function
Mean value	i16	int16_t	R_DSP_MeanAbs_i16(const vector_t * input, int16_t * mean)
	i32	int32_t	R_DSP_MeanAbs_i32(const vector_t * input, int32_t * mean)
	f32	float	R_DSP_MeanAbs_f32(const vector_t * input, float * mean)
Maximum value	i16	int16_t	R_DSP_MaxAbs_i16(const vector_t * input, int16_t * max)
	i32	int32_t	R_DSP_MaxAbs_i32(const vector_t * input, int32_t * max)
	f32	float	R_DSP_MaxAbs_f32(const vector_t * input, float * max)
Mean value and maximum value	i16	int16_t	R_DSP_MeanMaxAbs_i16(const vector_t * input, int16_t * mean, int16 * max)
	i32	int32_t	R_DSP_MeanMaxAbs_i32(const vector_t * input, int32_t * mean, int32 * max)
	f32	float	R_DSP_MeanMaxAbs_f32(const vector_t * input, float * mean, float * max)

Arguments

input	Pointer to a vector_t that stores the input data. The referenced members are as follows.
input->n	Input data count. It supports up to 65536.
input->data	Pointer to the first element of a real number array.
mean	Pointer to a variable that stores the mean value calculated by these functions.
max	Pointer to a variable that stores the maximum value calculated by these functions.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurs when absolute value is maximum value
R_DSP_ERR_INPUT_NULL	Pointer to input vector or data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is outside specifiable range.
R_DSP_ERR_OUTPUT_NULL	Pointer to output is null.

Description

These functions calculate the mean value and/or maximum value of the absolute values of the real numbers stored in an input array. The mean value of the absolute values is defined by the following formula.

$$\frac{1}{N}\sum_{i=0}^{N-1}|x_i|$$

Here, x_i represents the *i*th input sample, and N represents the sample count.

Example

```
#include <r_dsp_statistical.h>

float  array[65535];
float  avg_abs, peak_abs;
r_dsp_status_t   my_sts = R_DSP_STATUS_OK;

/* fill it up with something interesting ... */
vector_t v = {65535, array};

/* called separately */
my_sts = R_DSP_MeanAbs_f32(&v, &avg_abs);
my_sts = R_DSP_MaxAbs_f32(&v, &peak_abs);

/* called together */
my_sts = R_DSP_MeanMaxAbs_f32(&v, &avg_abs, &peak_abs);
```

4.7 Mean Value and Sample Variance Functions

These functions calculate the mean value and sample variance of a real number array.

Format and Function List

```
r_dsp_status_t R_DSP_MeanVar_<intype>(
   const vector_t * input,
   <outtype1> * mean,
   <outtype2> * variance
)
```

<intype> can be i16, i32, or f32.

<outtype1> must be an int16_t, int32_t, or float of the same type as <intype>.

<outtype2> is float if <intype> is f32, or int32_t if <intype> is i16 or i32.

Table 4.7 shows the functions for the respective combinations of <intype>, <outtype1>, and <outtype2>.

Table 4.7 Mean Value and Sample Variance Function List

<intype>	<outtype1>	<outtype2>	Function
i16	int16_t	int32_t	R_DSP_MeanVar_i16(const vector_t * input, int16_t * mean, int32_t * variance)
i32	int32_t	int32_t	R_DSP_MeanVar_i32(const vector_t * input, int32_t * mean, int32_t * variance)
f32	float	float	R_DSP_MeanVar_f32(const vector_t * input, float * mean, float * variance)

Arguments

input	Pointer to a vector_t that stores the input data. The referenced members are as follows.
input->n	Input data count. It supports up to 65536.
input->data	Pointer to the first element of a real number array.
mean	Pointer to a variable that stores the mean value calculated by these functions.
variance	Pointer to a variable that stores the sample variance value calculated by these functions.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Pointer to input vector or data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is outside specifiable range.
R_DSP_ERR_OUTPUT_NULL	Pointer to output is null.

Description

Sample variance is defined by the following formula.

$$\frac{1}{N}\sum_{i=0}^{N-1}(x_i - \bar{x})(x_i - \bar{x})$$

Here, x_i represents the *i*th input sample, N represents the input sample count, and $\overline{x}$ represents the mean value of the input samples.

Example

An example of calculating both the mean value and the sample variance at once is shown below.

```
#include <r_dsp_statistical.h>

float    array[65535];
float    mean;
float    var;

r_dsp_status_t   my_sts = R_DSP_STATUS_OK;

/* fill it up with something interesting ... */
vector_t   v = {65535, array};

/* call the library function */
my_sts = R_DSP_MeanVar_f32(&v, &mean, &var);
```

4.8 Sample Variance Functions

These functions calculate the sample variance from a real number array and a mean value.

Format and Function List

```
r_dsp_status_t R_DSP_Var_GivenMean_<intype1>(
   const vector_t * input,
   const <intype2> mean,
   <outtype> * variance
)
```

<intype1> can be i16, i32, or f32.

<intype2> must be an int16_t, int32_t, or float of the same type as <intype1>.

<outtype> is float if <intype1> is f32, or int32_t if <intype1> is i16 or i32.

Table 4.8 shows the functions for the respective combinations of <intype1>, <intype2>, and <outtype>.

Table 4.8 Sample Variance Function List

<intype1>	<intype2>	<outtype>	Function Name
i16	int16_t	int32_t	R_DSP_Var_GivenMean_i16(const vector_t * input, const int16_t mean, int32_t * variance)
i32	int32_t	int32_t	R_DSP_Var_GivenMean_i32(const vector_t * input, const int32_t mean, int32_t * variance)
f32	float	float	R_DSP_Var_GivenMean_f32(const vector_t * input, const float mean, float * variance)

Arguments

input	Pointer to a vector_t that stores the input data. The referenced members are as follows.
input->n	Input data count. It supports up to 65536.
input->data	Pointer to the first element of a real number array.
mean	Mean value of the input data.
variance	Pointer to a variable that stores the sample variance value calculated by these functions.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Pointer to input vector or data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is outside specifiable range.
R_DSP_ERR_OUTPUT_NULL	Pointer to output is null.

Description

These functions calculate the sample variance from a real number array and a mean value. Sample variance is defined by the following formula.

$$\frac{1}{N}\sum_{i=0}^{N-1}(x_i - \bar{x})(x_i - \bar{x})$$

Here, x_i represents the *i*th input sample of the input data array, and N represents the input sample count, and $\overline{x}$ represents the mean value of the input samples.

Example

```
#include <r_dsp_statistical.h>

int32_t   array[65535];
int32_t   mean = 0x0123FFFF; // assume we already have this
int32_t   var;

r_dsp_status_t   my_sts = R_DSP_STATUS_OK;

/* fill it up with something interesting ... */
vector_t v = {65535, array};

/* call a different library function */
my_sts = R_DSP_Var_GivenMean_i32(&v, mean, &var);
```

4.9 Histogram Functions

These functions calculate a histogram from a real number array.

Format and Function List

```
r_dsp_status_t R_DSP_Histogram_<intype1><outtype>(
   const vector_t * input,
   vector_t * histo,
   const <intype2> offset,
   const <intype2> scale
)
```

<intype1> can be i16, i32, or f32.

<outtype> is ui16, the data type of the histogram.

<intype2> must be an int16_t, int32_t, or float of the same type as <intype1>.

Table 4.9 shows the functions for the respective combinations of <intype1> and <intype2>.

Table 4.9 Histogram Function List

<intype1>	<intype2>	Function Name
i16	int16_t	R_DSP_Histogram_i16ui16(const vector_t * input, vector_t * histo, const int16_t offset, const int16_t scale)
i32	int32_t	R_DSP_Histogram_i32ui16(const vector_t * input, vector_t * histo, const int32_t offset, const int32_t scale)
f32	float	R_DSP_Histogram_f32ui16(const vector_t * input, vector_t * histo, const float offset, const float scale)

Arguments

input	Pointer to a vector_t that stores the input data. The referenced members are as follows.
input->n	Input data count.
input->data	Pointer to the first element of a real number array.
histo	Pointer to a vector_t that stores the frequency. The referenced members are as follows.
histo->n	Bin count of the histogram.
histto->data	Pointer to the first element of an unsigned 16-bit integer array.
offset	Offset value to apply to the input data.
scale	Scaling parameter of the input data values that have been offset. The scaling parameter must be 1.0 or lower. In order to represent 1.0, the format of the scaling parameter is Q2.14 format for i16, or Q2.30 format for i32.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Pointer to input vector or data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is 0 or histo->n is 0.
R_DSP_ERR_OUTPUT_NULL	Pointer to output is null.
R_DSP_STATUS_HISTO_OUT_OF_RANGE	Result, normalized based on scale and offset, is negative or greater than or equal to histo->n.

Description

A histogram is a discrete approximation of a probability distribution function (PDF) for the input samples. These functions normalize the input samples based on offset and scale, and, using the integer portion as an index, count the frequencies with which these fall within the respective regions (bins) on the histogram.

Normalization of the input samples is performed based on offset and scale, according to the following formula.

$$\tilde{x_i} = (x_i - offset) * scale$$

Here, x_i represents the *i*th input sample, and $\tilde{x_i}$ represents the *i*th normalized input sample.

For example, to find a histogram with 200 bins for input samples with a range of values of [-1000, 999], decide on an offset and a scale such that the range of values for the normalized input samples will be [0, 199].

Example

An example is shown below in which a histogram with 200 bins is found for 32-bit fixed-point input samples with a range of values of [-1000, 999]. In this example, the input samples are offset by 1,000 and reduced to 1/10, and, taking the input sample value range to be [0, 199], a 200-bin result is obtained.

```
#include <r_dsp_statistical.h>

int32_t      myarray[N];                    // input sample array
uint16_t     harray[200];           // histogram array

r_dsp_status_t   my_sts = R_DSP_STATUS_OK;

/*fill the array with something interesting… */
vector_t         inputVector = {N, myarray];
vector_t         histo = {200, harray};
const int32_t    DIVTEN = 0x06666666; //  1/10 in Q2.30 format

/* call the library function */
my_sts = R_DSP_Histogram_i32ui16(&inputVector, &histo, -1000, DIVTEN);
```

4.10 Mean Value and Mean Absolute Deviation (MAD) Functions

These functions calculate the mean value and mean absolute deviation (MAD) of a real number array.

Format and Function List

```
r_dsp_status_t R_DSP_MeanMAD_<intype>(
   const vector_t * input,
   <outtype> * mean,
   <outtype> * mad
)
```

<intype> can be i16, i32, or f32.

<outtype> must be an int16_t, int32_t, or float of the same type as <intype>.

Table 4.10 shows the functions for the respective combinations of <intype1>, <intype2>, and <outtype>.

Table 4.10 Mean Value and Mean Absolute Deviation Function List

<intype>	<outtype>	Function Name
i16	int16_t	R_DSP_MeanMAD_i16(const vector_t * input, int16_t * mean, int16_t * mad)
i32	int32_t	R_DSP_MeanMAD_i32(const vector_t * input, int32_t * mean, int32_t * mad)
f32	float	R_DSP_MeanMAD_f32(const vector_t * input, float * mean, float * mad)

Arguments

input	Pointer to a vector_t that stores the input data. The referenced members are as follows.
input->n	Input data count. It supports up to 65536.
input->data	Pointer to the first element of a real number array.
mean	Pointer to a variable that stores the mean value calculated by these functions.
mad	Pointer to a variable that stores the mean absolute deviation value calculated by these functions.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Pointer to input vector or data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is outside specifiable range.
R_DSP_ERR_OUTPUT_NULL	Pointer to output is null.

Description

The mean absolute deviation is the value resulting from taking the simple mean of the absolute values of the differences between the mean value and the respective samples, and is defined by the following formula.

$$MAD = \frac{1}{N}\sum_{i=0}^{N-1} D_i = \frac{1}{N}\sum_{i=0}^{N-1} |x_i - \bar{x}|$$

Here, x_i represents the *i*th input sample, N represents the input sample count, and $\overline{x}$ represents the mean value of the input samples.

Example

```
#include <r_dsp_statistical.h>

float  array[65535];
float  mad;
float  mean;
r_dsp_status_t   my_sts = R_DSP_STATUS_OK;

/* fill it up with something interesting ... */
vector_t v = {65535, array};

/* calculate both mean and MAD */
my_sts = R_DSP_MeanMAD_f32(&v, &mean, &mad);
```

4.11 Mean Absolute Deviation (MAD) Functions

Given a mean value, these functions calculate the mean absolute deviation of a real number array.

Format and Function List

```
r_dsp_status_t R_DSP_MAD_GivenMean_<intype1>(
   const vector_t * input,
   const <intype2> mean,
   <outtype> * mad
)
```

<intype1> can be i16, i32, or f32.

<intype2> must be an int16_t, int32_t, or float of the same type as <intype1>.

<outtype> must be an int16_t, int32_t, or float of the same type as <intype1>.

Table 4.11 shows the functions for the respective combinations of <intype1>, <intype2>, and <outtype>.

Table 4.11 Mean Absolute Deviation Function List

<intype1>	<intype2> <output>	Function
i16	int16_t	R_DSP_MAD_GivenMean_i16(const vector_t * input, const int16_t mean, int16_t * mad)
i32	int32_t	R_DSP_MAD_GivenMean_i32(const vector_t * input, const int32_t mean, int32_t * mad)
f32	float	R_DSP_MAD_GivenMean_f32(const vector_t * input, const float mean, float * mad)

Arguments

input	Pointer to a vector_t that stores the input data. The referenced members are as follows.
input->n	Input data count. It supports up to 65536.
input->data	Pointer to the first element of a real number array.
mean	Mean value of the input data.
mad	Pointer to a variable that stores the mean absolute deviation value calculated by these functions.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Pointer to input vector or data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is outside specifiable range.
R_DSP_ERR_OUTPUT_NULL	Pointer to output is null.

Description

The mean absolute deviation (MAD) is the value resulting from taking the simple mean of the absolute values of the differences between the mean value and the respective samples, and is defined by the following formula.

$$MAD = \frac{1}{N}\sum_{i=0}^{N-1} D_i = \frac{1}{N}\sum_{i=0}^{N-1} |x_i - \bar{x}|$$

Here, x_i represents the *i*th input sample, N represents the input sample count, and $\bar{x}$ represents the mean value of the input samples.

Example

```
#include <r_dsp_statistical.h>

float  mean, mad;
r_dsp_status_t   my_sts = R_DSP_STATUS_OK;

/* calculate the mean */
R_DSP_Mean_f32(&v, &mean);

/* calculate MAD with mean already known */
my_sts = R_DSP_MAD_GivenMean_f32(&v, mean, &mad);
```

4.12 Median Functions (Working Memory Not Required)

These functions calculate the median value of a real number array. Since internal calculation results are stored in the input array, these functions do not require working memory.

Format and Function List

```
r_dsp_status_t R_DSP_Median_InPlace_<intype>(
   vector_t * input,
   <outtype> * median
)
```

<intype> can be i16, i32, or f32.

<outtype> must be an int16_t, int32_t, or float of the same type as <intype>.

Table 4.12 shows the functions for the respective combinations of <intype1>, <intype2>, and <outtype>.

Table 4.12 Median Function List

<intype>	<outtype>	Function
i16	int16_t	R_DSP_Median_InPlace_i16(vector_t * input, int16_t * median)
i32	int32_t	R_DSP_Median_InPlace_i32(vector_t * input, int32_t * median)
f32	float	R_DSP_Median_InPlace_f32(vector_t * input, float * median)

Arguments

input	Pointer to a vector_t that stores the input data. The referenced members are as follows.
input->n	Input data count from 3 to 65536.
input->data	Pointer to the first element of a real number array. Since the input data is overwritten by these functions, the contents of the memory after calling the functions are undefined.
median	Pointer to a variable that stores the median value calculated by these functions.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Pointer to input vector is null, or data member of the vector is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is outside specifiable range.
R_DSP_ERR_OUTPUT_NULL	Pointer to output is null.

Description

These functions calculate the median value of a real number array. Since the input vector is used to store internal calculation results, these functions do not require working memory, but the input vector values after calling the functions are undefined.

The median value of the input samples is the sample that is in the center after sorting the samples. If the number of input samples is even number, the median is defined to be the average of the two samples in the center.

Example

```
#include <r_dsp_statistical.h>

float  array[6] = {1.0f, 6.0f, 2.0f, 8.0f, 7.0f, 2.0f};
   /* input buffer will be overwritten. */
float  median;
r_dsp_status_t   my_sts = R_DSP_STATUS_OK;   // place to return status
vector_t  inputArray = {6, array};

/* median will be 3.5 after the function call below */
my_sts = R_DSP_Median_InPlace_f32(&inputArray, &median);
```

4.13 Median Functions (Working Memory Used)

These functions calculate the median value of a real number array. These functions require working memory in order to store internal calculation results.

Format and Function List

```
r_dsp_status_t R_DSP_Median_<intype1>(
   const vector_t * input,
   <intype2> * scratch,
   <outtype> * median
)
```

<intype1> can be i16, i32, or f32.

<intype2> must be an int16_t, int32_t, or float of the same type as <intype1>.

<outtype> must be an int16_t, int32_t, or float of the same type as <intype1>.

Table 4.13 shows the functions for the respective combinations of <intype1>, <intype2>, and <outtype>.

Table 4.13 Median Function List

<intype1>	<intype2> <outtype>	Function Name
i16	int16_t	R_DSP_Median_i16(const vector_t * input, int16_t *scratch, int16_t * median)
i32	int32_t	R_DSP_Median_i32(const vector_t * input, int32_t *scratch, int32_t * median)
f32	float	R_DSP_Median_f32(const vector_t * input, float *scratch, float * median)

Arguments

input	Pointer to a vector_t that stores the input data. The referenced members are as follows.
input->n	Input data count from 3 to 65536.
input->data	Pointer to the first element of a real number array. The input data is not modified by these functions.
scratch	Pointer to the working memory used by the function. Working memory having a length of at least the same as the input data count is required. The contents of the memory after calling these functions are undefined.
median	Pointer to a variable that stores the median value calculated by these functions.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Pointer to input vector, data member, or working memory is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is outside specifiable range.
R_DSP_ERR_OUTPUT_NULL	Pointer to output is null.

Description

These functions calculate the median value of a real number array. These functions require working memory to store internal calculation results. The contents of the memory after these functions exit are undefined.

The median value of the input samples is the sample that is in the center after sorting the samples. If the number of input samples is even number, the median is defined to be the average of the two samples in the center.

Example

```
#include <r_dsp_statistical.h>

const float  array[6] = {1.0f, 6.0f, 2.0f, 8.0f, 7.0f, 2.0f};  //input won't change.
float     scratch[6];   // the same size as input
float     median;

r_dsp_status_t   my_sts = R_DSP_STATUS_OK;   // place to return status

vector_t  inputArray = {6, array};

/* median will be 3.5 after the function call below */
my_sts = R_DSP_Median_f32(&inputArray, scratch, &median);
```

5. Filter Operation API

This section describes the following filter function APIs, which are supported by the DSP Library.

- Generic FIR filter
- IIR Biqaud filter
- Single-Pole IIR filter

5.1 Generic FIR Filter

5.1.1 Overview

The generic FIR filter API provides FIR filter operations on each input sample.

The operation of the T-tap FIR filter is defined by the following formula.

$$y(n) = \sum_{i=0}^{T-1} h(i) * x(n-i)$$

Here, n is the sample index, $h(i)$ is the coefficient, $x(n-i)$ is the input data of the *i*th sample delay, and $y(n)$ is the output data.

Based on the above formula, the generic FIR filter operation implements the signal flow shown in Figure 5.1.

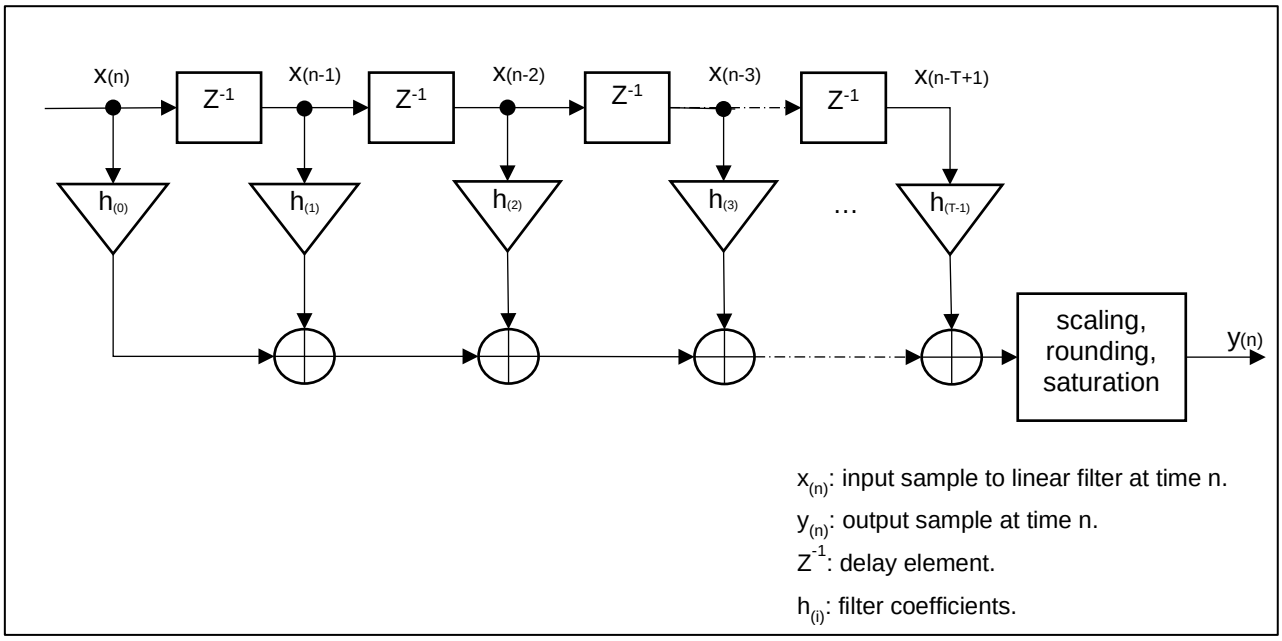


Figure 5.1 FIR Filter Signal Flow Chart

The generic FIR filter operates on an array in which input samples and delay samples are stored sequentially. A pointer to the array that stores the input samples and delay samples is stored in the handle described below, and the input sample count is specified by the input vector. The orders to store the data in the respective arrays are shown in Figure 5.2.

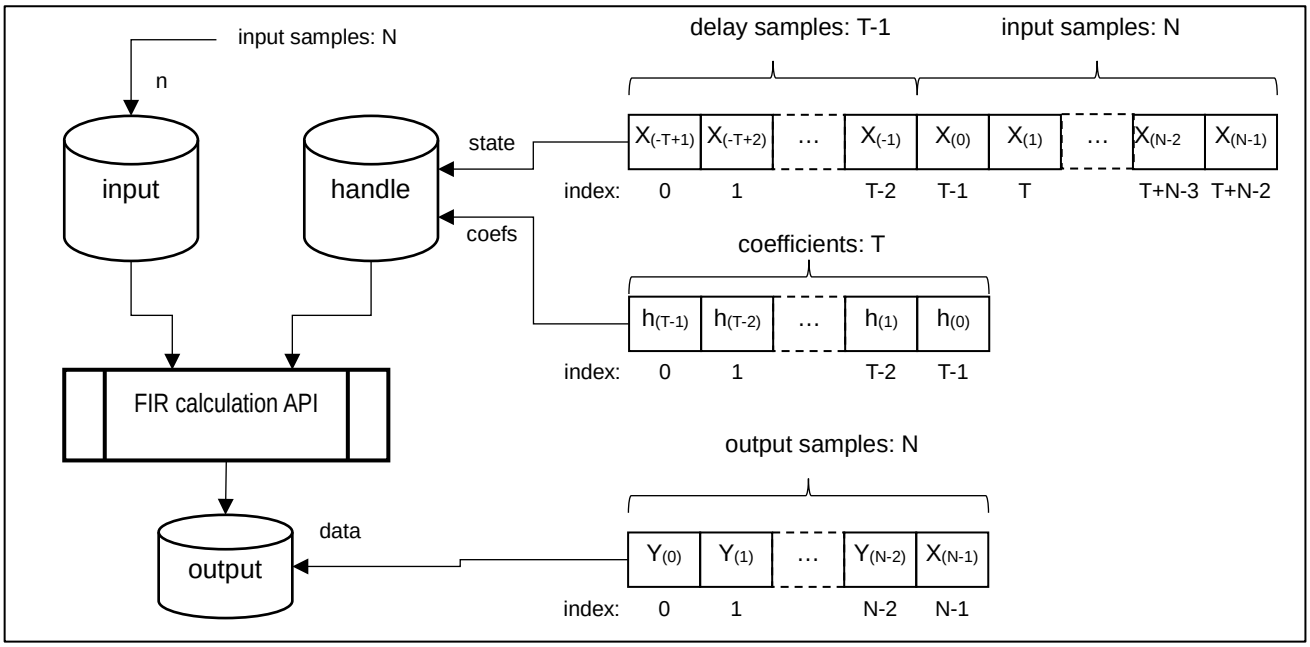


Figure 5.2 Data Storage Order in FIR Filter API

5.1.2 Generic FIR Filter API

The generic FIR filter API performs initialization and operation using r_dsp_firfilter_t handle. The handle gives information such as coefficients, operation conditions and a delay data array. The handle's data structure is described later. A list of the API functions is shown in Table 5.1, and the combinations of input and output types are shown in Table 5.2. The data type of the coefficients is the same as that of the input data type.

Table 5.1 Generic FIR Filter API Functions

API	Description
R_DSP_FIR_Init_<intype><outtype>	Performs filter initialization.
R_DSP_FIR_<intype><outtype>	Performs filter operation.

Table 5.2 Combination of Input and Output Data Types in Generic FIR Filter API

<intype>	<outtype>
i16	i16, i32
ci16	ci16, ci32
i32	i32
ci32	ci32
f32	f32
cf32	cf32

5.1.2.1 Generic FIR Handle Data Structure

The generic FIR filter API uses the r_dsp_firfilter_t handle data structure. The definition of the structure is shown below.

```
typedef struct
{
   uint32_t  taps;     // number of filter taps
   void *    coefs;    // pointer to filter coefficients
   void *    state;    /* pointer to filter state data, including the filter's delay
                       line and any other implementation-dependent state */
   scale_t   scale;    // scaling parameter
   uint32_t  options;  // options that specify rounding, saturation, or other behaviors
} r_dsp_firfilter_t;
```

The members of the handle data structure are described in Table 5.3.

Table 5.3 Descriptions of r_dsp_firfilter_t Structure Members

Member	Description
taps	Number of Filter taps. The specifiable range is 1 to 2^{16} = 65536.
coefs	Specifies a pointer to the coefficient array. The data type of the coefficients is the same as that of the input data. The coefficients are provided by the user. And those are stored in the order (h_{taps-1}, h_{taps-2}, ..., h_3, h_2, h_1, h_0).
state	Specifies a pointer to the beginning of the delay data array. In the array, the oldest delay data is stored at the beginning, and the delay data and input data are stored in the following order. ($X_{[-taps+1]}$, $X_{[-taps+2]}$, ..., $X_{[-3]}$, $X_{[-2]}$, $X_{[-1]}$, $X_{[0]}$, $X_{[1]}$, $X_{[2]}$, $X_{[3]}$, ..., $X_{[N-1]}$), N: input data count
scale	Output data scaling parameter. For details, see 2.9 Scaling. For fixed-point operations, the filter operation output is right-shifted corresponding to this value. The scaling parameter is an integer, and the valid value ranges are as follows. i32i32, ci32ci32 format: 1 to 62 i16i16, ci16ci16 format: 1 to 30 i16i32, ci16ci32 format: -31 to +31 (negative values indicate left-shifting) For floating-point operations, the filter operation output is multiplied by this value. The scaling parameter is a floating-point value. When the value is greater than 1.0, the results are amplified. And when the value is smaller than 1.0, the results are attenuated.
options	Operation control options. The fixed-point operation functions support rounding modes and saturation modes. The floating-point operation functions do not refer to them. The following two saturation modes are supported. R_DSP_NOSATURATE (default) R_DSP_SATURATE The following two rounding modes are supported. R_DSP_ROUNDING_TRUNC (default) R_DSP_ROUNDING_NEAREST

NOTE: Overflow may occur if the number of filter taps is 512 or more.

5.1.2.2 Generic FIR Filter Initialization Functions

These functions initialize the delay data of the generic FIR filter.
They refer to the handle members and initialize taps-1 elements of the delay data array to 0 from the first element. The array is specified by the state pointer.
Execute one of the functions once before executing the FIR filter operation API.

Format and Function List

```
r_dsp_status_t R_DSP_FIR_Init_<intype><outtype>(r_dsp_firfilter_t * handle)
```

Table 5.4 shows the functions for the respective combinations of <intype> and <outtype>.

Table 5.4 Generic FIR Filter Initialization Function List

<intype>	<outtype>	Function
i16	i16	R_DSP_FIR_Init_i16i16(r_dsp_firfilter_t * handle)
	i32	R_DSP_FIR_Init_i16i32(r_dsp_firfilter_t * handle)
ci16	ci16	R_DSP_FIR_Init_ci16ci16(r_dsp_firfilter_t * handle)
	ci32	R_DSP_FIR_Init_ci16ci32(r_dsp_firfilter_t * handle)
i32	i32	R_DSP_FIR_Init_i32i32(r_dsp_firfilter_t * handle)
ci32	ci32	R_DSP_FIR_Init_ci32ci32(r_dsp_firfilter_t * handle)
f32	f32	R_DSP_FIR_Init_f32f32(r_dsp_firfilter_t * handle)
cf32	cf32	R_DSP_FIR_Init_cf32cf32(r_dsp_firfilter_t * handle)

Arguments

handle	Pointer to an r_dsp_firfilter_t data structure. The following members of the structure are referred to. For details, see Table 5.3.
handle->taps	Filter tap count.
handle->state	Pointer to a delay data array.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_HANDLE_NULL	Pointer to handle is null.
R_DSP_ERR_STATE_NULL	Pointer to state is null.
R_DSP_ERR_INVALID_TAPS	Number of filter taps is outside specifiable range.

5.1.2.3 Generic FIR Filter Operation Functions

These functions carry out FIR filter operation with the number of operations equal to the input sample count, and output the results.

Format and Function List

```
r_dsp_status_t R_DSP_FIR_<intype><outtype>(
   const r_dsp_firfilter_t * handle,
   const vector_t * input,
   vector_t * output
)
```

Table 5.5 shows the functions for the respective combinations of <intype> and <outtype>.

Table 5.5 Generic FIR Filter Operation Function List

<intype>	<outtype>	Function
i16	i16	R_DSP_FIR_i16i16(r_dsp_firfilter_t * handle, const vector_t * input, vector_t * output)
	i32	R_DSP_FIR_i16i32(r_dsp_firfilter_t * handle, const vector_t * input, vector_t * output)
ci16	ci16	R_DSP_FIR_ci16ci16(r_dsp_firfilter_t * handle, const vector_t * input, vector_t * output)
	ci32	R_DSP_FIR_ci16ci32(r_dsp_firfilter_t * handle, const vector_t * input, vector_t * output)
i32	i32	R_DSP_FIR_i32i32(r_dsp_firfilter_t * handle, const vector_t * input, vector_t * output)
ci32	ci32	R_DSP_FIR_ci32ci32(r_dsp_firfilter_t * handle, const vector_t * input, vector_t * output)
f32	f32	R_DSP_FIR_f32f32(r_dsp_firfilter_t * handle, const vector_t * input, vector_t * output)
cf32	cf32	R_DSP_FIR_cf32cf32(r_dsp_firfilter_t * handle, const vector_t * input, vector_t * output)

Arguments

handle	Pointer to an r_dsp_firfilter_t data structure. All members of the structure are referred to. For details, see Table 5.3. The input data is stored in an array to which the member “state” has been set.
input	Pointer to the vector_t to input to the filter. The following member is referred to.
input->n	Input data count.
output	Pointer to the vector_t that stores the filter output. The following members are referred to.
output->n	Number of elements in the array to which the data member points. Must be greater than or equal to the input data count.
output->data	Pointer to the beginning of the array that stores the filter output.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with "_Check").
R_DSP_ERR_HANDLE_NULL	Pointer to handle is null.
R_DSP_ERR_INPUT_NULL	Input vector or its data member is null.
R_DSP_ERR_OUTPUT_NULL	Output vector or its data member is null.
R_DSP_ERR_STATE_NULL	member “state” of the handle is null.
R_DSP_ERR_COEFF_NULL	coefs member of the handle is null.
R_DSP_ERR_INVALID_TAPS	Number of taps is outside specifiable range.
R_DSP_ERR_INVALID_SCALE	Value of scale member of handle is outside specifiable range.
R_DSP_ERR_INVALID_OUTPUT_SIZE	Output buffer size is too small.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is 0.

NOTE: When specifying 512 or greater as Filter tap count, overflow could occur as in R_DSP_FIR_i32i32 and R_DSP_FIR_ci32ci32 operation. Therefore, the coefficients should be designed to avoid it.

5.1.3 Example

An example of the initialization and operation of a 64-tap, 16-bit fixed-point real number FIR filter is shown below. The

other data formats are analogous.

```
#include <r_dsp_filters.h>
#define NUM_TAPS (64)     // FIR Filter taps
#define NUM_SAMPLES (16)  // input samples

r_dsp_firfilter_t  myFilterHandle;    // instantiate a handle for this filter
vector_t           myInput;           // input vector
vector_t           myOutput;          // output vector

/* Coefficients should be stored in reversed order */
int16_t            myCoeffs[NUM_TAPS] = {…}; //{hT-1,hT-2,…,h2,h1,h0}

/* The input data buffer should contain previous (T-1) input samples (i.e. delay line)
* contiguous with the present (N) input samples
* {X[-T+1],X[-T+2],…,X[-2],X[-1],X[0],X[1],X[2],…,X[N-1]}
*/
int16_t            inputData[NUM_TAPS - 1 + NUM_SAMPLES];

/* The output data will be stored in time-sequential order */
int16_t            outputData[NUM_SAMPLES];

r_dsp_status_t myFIRFlags = R_DSP_STATUS_OK; // place to store return status

/*-------------- Set up the FIR filter ------------*/
myFilterHandle.taps = NUM_TAPS;  // filter taps
myFilterHandle.scale.i32 = 15;   // the scaling factor for output data to Q1.15 format
myFilterHandle.options = 0;      // default
myFilterHandle.state = (void *)&inputData[0]; // starting address of delayline
myFilterHandle.coefs = (void *)myCoeffs;    // starting address of coefficnts array

/*----- Initialize the coefficients and internal state ------*/
myFIRFlags = R_DSP_FIR_Init_i16i16(&myFilterHandle);

/*---------------- Set up the input/output ----------------*/
myInput.n = NUM_SAMPLES;
myOutput.n = NUM_SAMPLES;
myOutput.data = (void *)outputData;

/*------------------ Wait for input data ------------------*/

/*--------------- Main library function call --------------*/
myFIRFlags = R_DSP_FIR_i16i16(&myFilterHandle, &myInput, &myOutput);

/*--------------- Output data are now ready ---------------
* Note: At this point myOutput.n holds the number of output samples generated by
* the library, where the data are written to the array pointed to by myOutput.data.
*-------------------------------------------------------------*/
```

5.2 IIR Biquad Filter

5.2.1 Overview

The IIR biquad filter API provides cascaded biquad-form second-order infinite impulse response (IIR) filter operations. The biquad filter has two forms, direct form I and direct form II, which can be selected by the user. By default, direct form I is selected for fixed-point operations, and direct form II is selected for floating-point operations.

The biquad-form second-order IIR M-stage cascaded transfer function is as shown in the formula below.

$$H(z) = \prod_{m=1}^{M} \frac{b_0^m + b_1^m * Z^{-1} + b_2^m * Z^{-2}}{1 + a_1^m * Z^{-1} + a_2^m * Z^{-2}}$$

Here, b_0^m, b_1^m, b_2^m are the feedforward coefficients, and a_1^m, a_2^m are the feedback coefficients.

The operation of the biquad-form second-order IIR filters is defined by the following formula.

$$y(n) = b_0 * x(n) + b_1 * x(n-1) + b_2 * x(n-2) - a_1 * y(n-1) - a_2 * y(n-2)$$

Here, y(n) represents the filter output, x(n) represents the input samples, and x(n-1) and y(n-1), respectively, represent the input data and output data delayed by one sampling period.

This API provides biquad-form second-order IIR filters in both direct form I and direct form II. The signal flow charts implemented for these respective forms are shown below.

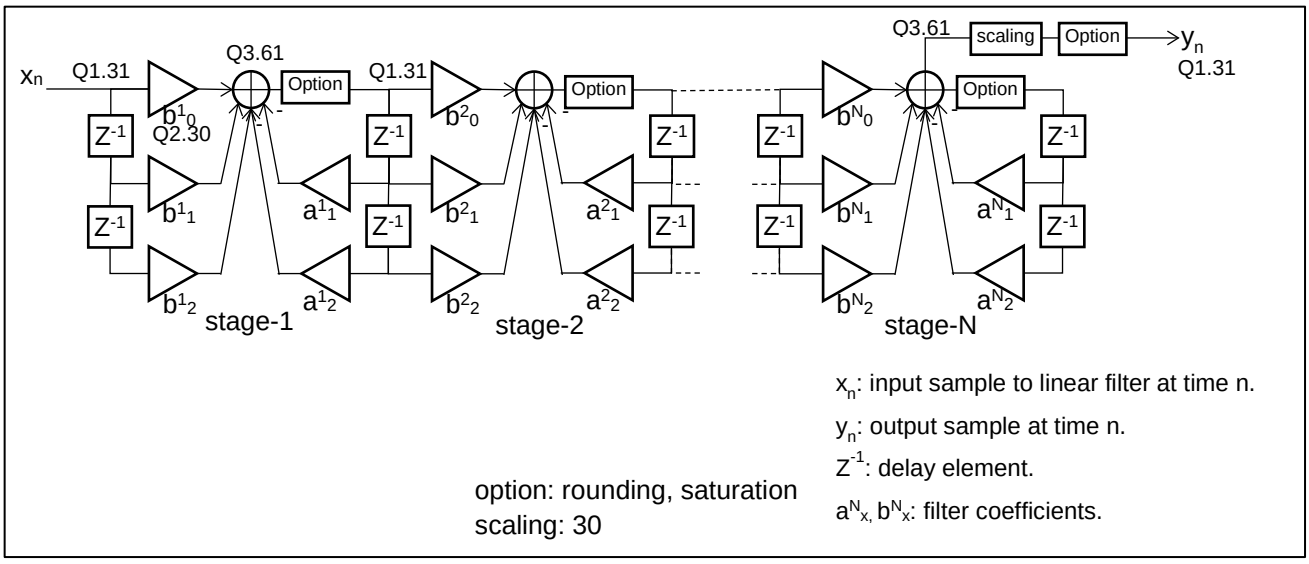


Figure 5.3 Signal Flow Chart for Implementation of IIR Biquad Standard Form I (Fixed-Point: Input Q1.31, Coefficients Q2.30)

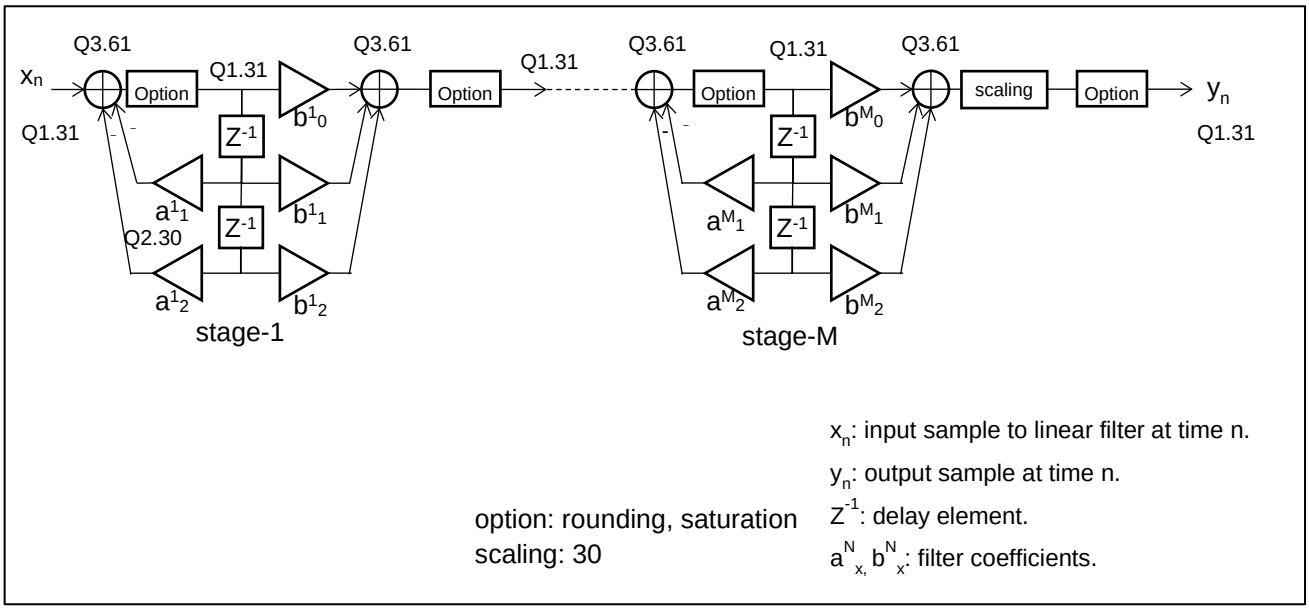


Figure 5.4 Signal Flow Chart for Implementation of IIR Biquad Standard Form II (Fixed-Point: Input Q1.31, Coefficients Q2.30)

5.2.2 IIR Biquad Filter API

The IIR biquad filter API performs initialization and operation using the r_dsp_iirbiquad_t handle. The handle gives information such as coefficients, operation conditions and a delay data array. The handle's data structure is described later.

A list of the API functions is shown in Table 5.6, and the combinations of input and output types are shown in Table 5.7. The data type of the coefficients is the same as that of the input data type.

Table 5.6 IIR Biquad Filter API Functions

API	Description
R_DSP_IIRBiquad_StateSize_<intype><outtype>	Returns the size required for the delay data array of the filter.
R_DSP_IIRBiquad_Init_<intype><outtype>	Performs filter initialization.
R_DSP_IIRBiquad_<intype><outtype>	Performs filter operation.

Table 5.7 Combination of Input and Output Data Types in IIR Biquad Filter API

<intype>	<outtype>
i16	i16, i32
ci16	ci16, ci32
i32	i32
ci32	ci32
f32	f32
cf32	cf32

5.2.2.1 IIR Biquad Handle Data Structure

The biquad IIR filter API uses the r_dsp_iirbiquad_t handle data structure. The definition of the structure is shown below.

```
typedef enum {
    R_DSP_BIQUAD_FORM_DEFAULT = 0,
    R_DSP_BIQUAD_FORM_I = 1,
    R_DSP_BIQUAD_FORM_II
} r_dsp_iirbiquad_form_t;

typedef struct
{
    uint32_t   stages;    // number of biquad stages
    void *     coefs;     // pointer to filter coefficients
    void *     state;     // pointer to filter’s delay line
    scale_t    scale;     // scaling factor
    uint32_t   qint;      // number of integer bit (0 or 1) for coefficients
    uint32_t   options; // options that specify rounding, saturation, or other behaviors
    uint32_t   form;      // which biquad form to use
} r_dsp_iirbiquad_t;
```

The members of the handle data structure are described in Table 5.8.

Table 5.8 Descriptions of r_dsp_iirbiquad_t Structure Members

Member	Description
stages	Stage count of the biquad filter 1 to 2^{27}-2 (=134217726)
coefs	Pointer to the coefficient array. The data type of the coefficients is the same as that of the input data. And the coefficients are provided by the user. The coefficients for each stage are stored in the order (b_0, b_1, b_2, a_1, a_2).
state	Pointer to the delay data array. The delay data array is defined by the user and it is maintained by the respective functions.
scale	Output data scaling parameter. For details, see 2.9 Scaling. For fixed-point operations, the filter operation output is right-shifted corresponding to this value. The scaling parameter is an integer, and the valid value ranges are as follows. i32i32, ci32ci32 format: 1 to 62 i16i16, ci16ci16 format: 1 to 30 i16i32, ci16ci32 format: -31 to +31 (negative values indicate left-shifting) For floating-point operations, the filter operation output is multiplied by this value. The scaling parameter is a floating-point value. When the value is greater than 1.0, the results are amplified. And when the value is smaller than 1.0, the results are attenuated.
qint	The number of integer bits, in Q format, of the filter coefficients for the fixed-point functions. For a value of 1.0 or less, the IIR coefficients do not have integer bits, and for a value of greater than 1.0 (and less than 2.0), they require one integer bit. As a result, the following two values can be specified for qint. 0: The number of integer bits, not including the sign bit, is 0. Example: Q1.15, Q1.31 1: The number of integer bits, not including the sign bit, is 1. Example: Q2.14, Q2.30 The floating-point functions do not refer to them.
options	Operation control options. The fixed-point operation functions support rounding modes and saturation modes. The floating-point operation functions do not refer to them. The following two saturation modes are supported. R_DSP_NOSATURATE (default) R_DSP_SATURATE The following two rounding modes are supported. R_DSP_ROUNDING_TRUNC (default) R_DSP_ROUNDING_NEAREST
form	Biquad form. Three types can be selected. R_DSP_BIQUAD_FORM_DEFAULT: The form is not specified (and the defaults below are selected) R_DSP_BIQUAD_FORM_I: Direct form I (the default for fixed-point) R_DSP_BIQUAD_FORM_II: Direct form II (the default for floating-point)

5.2.2.2 IIR Biquad Filter Delay Data Array Size Acquisition Functions

These functions return the memory size of the delay data array required for the IIR biquad filter. The return values are shown in bytes. The memory size is calculated according to the filter stages specified in “stages” and the filter form specified in “form” in reference to the handle.

Format and Function List

```
int32_t R_DSP_IIRBiquad_StateSize_<intype><outtype>(const r_dsp_iirbiquad_t * handle)
```

Table 5.9 shows the functions for the respective combinations of <intype> and <outtype>.

Table 5.9 IIR Biquad Filter State Size Acquisition Function List

<intype>	<outtype>	Function
i16	i16	R_DSP_IIRBiquad_StateSize_i16i16(const r_dsp_iirbiquad_t * handle)
	i32	R_DSP_IIRBiquad_StateSize_i16i32(const r_dsp_iirbiquad_t * handle)
ci16	ci16	R_DSP_IIRBiquad_StateSize_ci16ci16(const r_dsp_iirbiquad_t * handle)
	ci32	R_DSP_IIRBiquad_StateSize_ci16ci32(const r_dsp_iirbiquad_t * handle)
i32	i32	R_DSP_IIRBiquad_StateSize_i32i32(const r_dsp_iirbiquad_t * handle)
ci32	ci32	R_DSP_IIRBiquad_StateSize_ci32ci32(const r_dsp_iirbiquad_t * handle)
f32	f32	R_DSP_IIRBiquad_StateSize_f32f32(const r_dsp_iirbiquad_t * handle)
cf32	cf32	R_DSP_IIRBiquad_StateSize_cf32cf32(const r_dsp_iirbiquad_t * handle)

Arguments

handle	Pointer to an r_dsp_iirbiquad_t data structure. The following members are referred to. For details, see Table 5.8.
handle->stages	Stage count of the filter.
handle->form	Filter form specification.

Return Values

These functions return the memory size, in bytes, of the delay data array that is required based on the conditions specified by the handle. Negative values represent the following errors.

R_DSP_ERR_HANDLE_NULL	Pointer to handle pointer is null.
R_DSP_ERR_INVALID_STAGES	Number of stages is outside specifiable range.

5.2.2.3 IIR Biquad Filter Initialization Functions

These functions initialize the delay data of the IIR biquad filter.
They refer to the handle members and initialize the delay data array to 0 according to the filter stage count "stages" and the filter form specification "form". The delay array is specified by the state pointer.
Execute one of the functions once before executing the IIR biquad filter operation API.

Format and Function List

```
r_dsp_status_t R_DSP_IIRBiquad_Init_<intype><outtype>(r_dsp_iirbiquad_t * handle)
```

Table 5.10 shows the functions for the respective combinations of <intype> and <outtype>.

Table 5.10 IIR Biquad Filter Initialization Function List

<intype>	<outtype>	Function
i16	i16	R_DSP_IIRBiquad_Init_i16i16(r_dsp_iirbiquad_t * handle)
	i32	R_DSP_IIRBiquad_Init_i16i32(r_dsp_iirbiquad_t * handle)
ci16	ci16	R_DSP_IIRBiquad_Init_ci16ci16(r_dsp_iirbiquad_t * handle)
	ci32	R_DSP_IIRBiquad_Init_ci16ci32(r_dsp_iirbiquad_t * handle)
i32	i32	R_DSP_IIRBiquad_Init_i32i32(r_dsp_iirbiquad_t * handle)
ci32	ci32	R_DSP_IIRBiquad_Init_ci32ci32(r_dsp_iirbiquad_t * handle)
f32	f32	R_DSP_IIRBiquad_Init_f32f32(r_dsp_iirbiquad_t * handle)
cf32	cf32	R_DSP_IIRBiquad_Init_cf32cf32(r_dsp_iirbiquad_t * handle)

Arguments

handle	Pointer to an r_dsp_iirbiquad_t data structure. The following members are referred to.
handle->stages	Stage count of the filter.
handle->form	Filter form specification.
handle->state	Pointer to a delay data array.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_HANDLE_NULL	Pointer to handle pointer is null.
R_DSP_ERR_STATE_NULL	State pointer is null.
R_DSP_ERR_INVALID_STAGES	Number of stages is outside specifiable range.

5.2.2.4 IIR Biquad Filter Operation Functions

These functions perform IIR biquad filter operations.
They refer to the handle members and perform the filter operation to a cascaded IIR biquad filter with the number of stages specified in "stages" by applying the coefficients specified in "coeffs".

Format and Function List

```
r_dsp_status_t R_DSP_IIRBiquad_<intype><outtype>(
   const r_dsp_iirbiquad_t * handle,
   const vector_t * input,
   vector_t * output
)
```

Table 5.11 shows the functions for the respective combinations of <intype> and <outtype>.

Table 5.11 IIR Biquad Filter Operation Function List

<intype>	<outtype>	Function
i16	i16	R_DSP_IIRBiquad_i16i16(const r_dsp_iirbiquad_t * handle, const vector_t * input, vector_t * output)
	i32	R_DSP_IIRBiquad_i16i32(const r_dsp_iirbiquad_t * handle, const vector_t * input, vector_t * output)
ci16	ci16	R_DSP_IIRBiquad_ci16ci16(const r_dsp_iirbiquad_t * handle, const vector_t * input, vector_t * output)
	ci32	R_DSP_IIRBiquad_ci16ci32(const r_dsp_iirbiquad_t * handle, const vector_t * input, vector_t * output)
i32	i32	R_DSP_IIRBiquad_i32i32(const r_dsp_iirbiquad_t * handle, const vector_t * input, vector_t * output)
ci32	ci32	R_DSP_IIRBiquad_ci32ci32(const r_dsp_iirbiquad_t * handle, const vector_t * input, vector_t * output)
f32	f32	R_DSP_IIRBiquad_f32f32(const r_dsp_iirbiquad_t * handle, const vector_t * input, vector_t * output)
cf32	cf32	R_DSP_IIRBiquad_cf32cf32(const r_dsp_iirbiquad_t * handle, const vector_t * input, vector_t * output)

Arguments

handle	Pointer to an r_dsp_iirbiquad_t data structure. All members of the structure are referred to. For details, see Table 5.8.
input	Pointer to the vector_t to input to the filter. The following members are referred to.
input->n	Input data count.
input->data	Pointer to the beginning of an array that stores the input data.
output	Pointer to the vector_t that stores the filter output. The following members are referred to.
output->n	Number of elements in the array to which the data member points. Must be greater than or equal to the input data count.
output->data	Pointer to the beginning of the array that stores the filter output.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with "_Check").
R_DSP_ERR_HANDLE_NULL	Pointer to handle pointer is null.
R_DSP_ERR_STATE_NULL	state member of the handle is null.
R_DSP_ERR_COEFF_NULL	coefs member of the handle is null.
R_DSP_ERR_INPUT_NULL	Input vector or data member is null.
R_DSP_ERR_OUTPUT_NULL	Output vector or data member is null.
R_DSP_ERR_INVALID_STAGES	Stage count is outside specifiable range.
R_DSP_ERR_INVALID_SCALE	Value of member “scale” of handle is outside specifiable range.
R_DSP_ERR_INVALID_OUTPUT_SIZE	Output buffer size is too small.
R_DSP_ERR_INVALID_INPUT_SIZE	Input sample count is 0.
R_DSP_ERR_INVALID_QINT	In case of fixed-point function, the specified number of integer bits is outside specifiable range.

5.2.3 Example

An example of the initialization and operation of a 3-band, 32-bit fixed-point real number IIR biquad filter is shown below.

```
#include <r_dsp_filters.h>
#define NUM_TAPS_PER_BIQUAD   (5)
#define NUM_BIQUAD_STAGES     (3)
#define NUM_SAMPLES           (16)
r_dsp_iirbiquad_t  myFilterHandle;    // instantiate a handle for my use
vector_t           myInput;           // input vector
vector_t           myOutput;          // output vector
/* Coefficients should be stored in reserved order as follows,
 1st band of b0, b1, b2, a1, a2,
 2nd band of b0, b1, b2, a1, a2,
 3rd band of b0, b1, b2, a1, a2, */
int32_t myCoeffs[NUM_TAPS_PER_BIQUAD * NUM_BIQUAD_STAGES] = {…};
int32_t                   myDLine[NUM_TAPS_PER_BIQUAD * NUM_BIQUAD_STAGES];
int32_t                   inputData[NUM_SAMPLES];
int32_t                   outputData[NUM_SAMPLES];
int32_t                   dynMemSize, staMemSize;
r_dsp_status_t     myIIRFlags = R_DSP_STATUS_OK; // place to store return status

/*-------------- Set up the IIR biquad filter ------------*/
myFilterHandle.stages = NUM_BIQUAD_STAGES;
myFilterHandle.form = R_DSP_BIQUAD_FORM_I;
myFilterHandle.scale.i32 = 30; // the scale factor for output data to Q1.31 format
myFilterHandle.qint = 1;       // coefficients and input all in Q2.30
myFilterHandle.options = 0;    // default
myFilterHandle.coefs = (void *)myCoeffs;

/* !!! It is important to setup the stages and the form before */
/* !!! calling function R_DSP_IIRBiquad_StateSize_xyyxyy() */
staMemSize = NUM_TAPS_PER_BIQUAD * NUM_BIQUAD_STAGES * sizeof(int32_t);
dynMemSize = R_DSP_IIRBiquad_StateSize_i32i32(&myFilterHandle);
if (staMemSize >= dynMemSize)
{
    myFilterHandle.state = (void *)myDLine;
}
else
{
    while(1);  // state size is not enough.
}

/* Initialize the coefficients and internal state */
myIIRFlags = R_DSP_IIRBiquad_Init_i32i32(&myFilterHandle);

/*---------------- Set up the input/output ----------------*/
myInput.n = NUM_SAMPLES;
myOutput.n = NUM_SAMPLES;
myInput.data = (void *)inputData;
myOutput.data = (void *)outputData;

/*------------------ Wait for input data ------------------*/
```

```
/*--------------- Main library function call --------------*/
myIIRFlags = R_DSP_IIRBiquad_i32i32(&myFilterHandle, &myInput, &myOutput);

/*--------------- Output data are now ready ---------------*/
/* Note: At this point myOutput.n holds the number of output samples generated by
* the library, where the data are written to the array pointed to by myOutput.data.
*-----------------------------------------------------------*/
```

5.3 Single-Pole IIR Filter

5.3.1 Overview

The single-pole IIR filter API provides low-pass and high-pass filter operations based on a first order infinite impulse response (IIR) filter taking real number input.

The signal flow of the filter is shown in Figure 5.5.

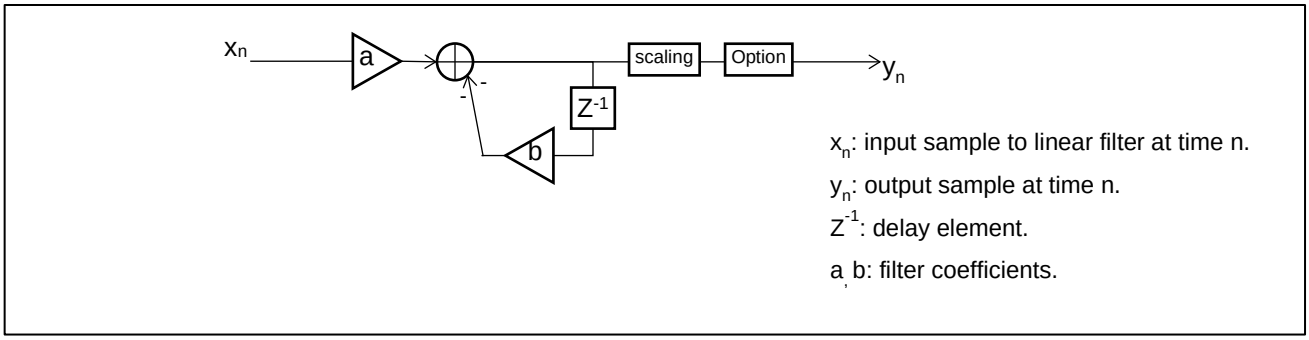


Figure 5.5 Single-Pole IIR Filter Signal Flow Chart

The transfer function of the single-pole IIR filter is defined by the following formula.

$$H(z) = \frac{a}{1 - b \cdot Z^{-1}}$$

Here, a and b represent the coefficients, and Z^{-1} represents the output primary delay data.

The single-pole IIR filter is a low-pass filter or a high-pass filter, depending on the coefficients.

Low-Pass Filter

If coefficient a is set to a positive value less than 1.0, by applying the coefficients to the following formula, the single-pole IIR filter becomes a low-pass filter.

$$H(z) = \frac{a}{1 - b \cdot Z^{-1}} = \frac{a}{1 - (1 - a) \cdot Z^{-1}}, \quad b = 1 - a, \qquad 0 < a < 1.0$$

The output of the single-pole IIR filter is calculated according to the following formula.

$$y_n = ax_n + by_{n-1} = ax_n + (1 - a)y_{n-1}$$

Here, x_n is the input sample, y_n is the output sample, and y_{n-1} is the primary delay of the output sample.

High-Pass Filter

If coefficient a is set to a negative value larger than -1.0, by applying the coefficients to the following formula, the single-pole IIR filter becomes a high-pass filter.

$$H(z) = \frac{a}{1 - b \cdot Z^{-1}} = \frac{a}{1 - (-1 - a) \cdot Z^{-1}}, \quad b = -1 - a, \qquad -1.0 < a < 0$$

The output of the single-pole IIR filter is calculated according to the following formula.

$$y_n = ax_n + by_{n-1} = ax_n + (-1 - a)y_{n-1}$$

Here, x_n is the input sample, y_n is the output sample, and y_{n-1} is the primary delay of the output sample.

5.3.2 Single-Pole IIR Filter API

The single-pole IIR filter API performs the operation using a r_dsp_iirsinglepole_t type handle. The handle gives information such as parameters, operation conditions and delay data arrays. The handle's data structure is described later. A list of the API functions is shown in Table 5.12, and the combinations of input and output types are shown in Table 5.13. The data format of the coefficients is the same as the input data format.

Table 5.12 Single-Pole IIR Filter API Function

API	Description
R_DSP_IIRSinglePole_<intype><outtype>	Performs filter operation.

Table 5.13 Combination of Input and Output Data Types in Single-Pole IIR Filter API

<intype>	<outtype>
i16	i16, i32
i32	i32
f32	f32

5.3.2.1 Single-Pole Filter Handle Data Structure

The single-pole IIR filter API uses a r_dsp_iirsinglepole_t handle data structure. The definition of the structure is shown below.

```
typdef struct
{
   void *    coefs;    // pointer to filter coefficient
   void *    state;    // pointer to filter's delay line
   scale_t   scale;    // scaling parameter
   uint32_t  options;  // options that specify rounding, saturation, or other behaviors
} r_dsp_iirsinglepole_t;
```

The members of the handle data structure are described in Table 5.14.

Table 5.14 Descriptions of r_dsp_iirsinglepole_t Structure Members

Member	Description
coefs	Pointer to the feedforward-side coefficient. The coefficient is provided by the user. If the coefficient is positive, the filter will be a low-pass filter, and if the coefficient is negative, the filter will be a high-pass filter. The range is -1.0 < coefficient < 1.0 with the exception of 0. The coefficient is Q1.31 format for the 32-bit fixed-point functions, and Q1.15 format for the 16-bit fixed-point functions.
state	Pointer to the delay data. The data type of the delay data is the same as that of the input data. The memory is provided by the user, and must be initialized before calling the operation functions. The delay data is maintained by the operation functions.
scale	Output data scaling parameter. For details, see 2.9 Scaling. For fixed-point operations, the filter operation output is right-shifted based on this value. The scaling parameter is an integer, and the valid value ranges are as follows. i32i32 format: 1 to 62 i16i16 format: 1 to 30 i16i32 format: -31 to +31 (negative values indicate left-shifting) For floating-point operations, the filter operation output is multiplied by this value. The scaling parameter is a floating-point value. When the value is greater than 1.0, the results are amplified. And when the value is smaller than 1.0, the results are attenuated.
options	Operation control options. The fixed-point operation functions support a rounding mode and a saturation mode. The floating-point operation functions do not refer to them. The following two saturation modes are supported. R_DSP_NOSATURATE (default) R_DSP_SATURATE The following two rounding modes are supported. R_DSP_ROUNDING_TRUNC (default) R_DSP_ROUNDING_NEAREST

5.3.2.2 Single-Pole IIR Filter Operation Functions

These functions perform single-pole IIR filter operations. They perform filter operations with the number of operations equal to the input data count, and output the results.

Format and Function List

```
r_dsp_status_t R_DSP_IIRSinglePole_<intype><outtype>(
   const r_dsp_iirsinglepole_t * handle,
   const vector_t * input,
   vector_t * output
)
```

Table 5.15 shows the functions for the respective combinations of <intype> and <outtype>.

Table 5.15 Single-Pole IIR Filter Operation Function List

<intype>	<outtype>	Function
i16	i16	R_DSP_IIRSinglePole_i16i16(r_dsp_iirsinglepole_t * handle, const vector_t * input, vector_t * output)
	i32	R_DSP_IIRSinglePole_i16i32(r_dsp_iirsinglepole_t * handle, const vector_t * input, vector_t * output)
i32	i32	R_DSP_IIRSinglePole_i32i32(r_dsp_iirsinglepole_t * handle, const vector_t * input, vector_t * output)
f32	f32	R_DSP_IIRSinglePole_f32f32(r_dsp_iirsinglepole_t * handle, const vector_t * input, vector_t * output)

Arguments

handle	Pointer to an r_dsp_iirsinglepole_t data structure. All members of the structure are referred to. For details, see Table 5.14.
input	Pointer to the vector_t to input to the filter. The following members are referred to.
input->n	Input data count.
input->data	Pointer to the beginning of an array that stores the input data.
output	Pointer to the vector_t that stores the filter output. The following members are referred to.
output->n	Number of elements in the array to which the data member points. Must be greater than or equal to the input data count.
output->data	Pointer to the beginning of the array that stores the filter output.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with "_Check").
R_DSP_ERR_HANDLE_NULL	Pointer to handle is null.
R_DSP_ERR_INPUT_NULL	Input vector or its data member is null.
R_DSP_ERR_OUTPUT_NULL	Output vector or its data member is null.
R_DSP_ERR_STATE_NULL	state member of the handle is null.
R_DSP_ERR_COEFF_NULL	coefs member of the handle is null.
R_DSP_ERR_INVALID_SCALE	Value of scale member of handle is outside specifiable range.
R_DSP_ERR_INVALID_OUTPUT_SIZE	Output buffer size is too small.
R_DSP_ERR_INVALID_INPUT_SIZE	Input sample count is 0.
R_DSP_ERR_INVALID_COEFF	Coefficients are outside specifiable range.

5.3.3 Example

An example of the use of a single-pole IIR filter function with a 32-bit fixed-point real number is shown below. The same mechanism is applied on other data formats supported by these functions.

```
#include <r_dsp_filters.h>
#define NUM_SAMPLES (16)

r_dsp_iirsinglepole_t  myHandle;
vector_t     myInput;  // See introduction section describing the API document
vector_t     myOutput; // for a definition of the “vector_t” data type.
int32_t      inputData[NUM_SAMPLES];
int32_t      outputData[NUM_SAMPLES];
int32_t      mystate;
int32_t      mycoeff;
r_dsp_status_t   my_sts
;
/*-------------- Set up the single-pole IIR filter ------------*/
mystate = 0;        // initialize state
mycoeff = (int32_t) (-0.15 * 0x7FFFFFFF);
myHandle.coefs = (void *)&mycoeff;
myHandle.state = (void *)&mystate;
myHandle.scale.i32 = 31;     // the scaling parameter for output data
myHandle.options = R_DSP_NOSATURATE | R_DSP_ROUNDING_TRUNC;

/*---------------- Set up the input/output ----------------*/
myInput.n = NUM_SAMPLES;
myOutput.n = NUM_SAMPLES;
myInput.data = (void *)inputData;
myOutput.data = (void *)outputData;

/*------------------ Wait for input data ------------------*/
/*--------------- Main library function call --------------*/
my_sts = R_DSP_IIRSinglePole_i32i32(&myHandle, &myInput, &myOutput);

/*--------------- Output data are now ready ---------------*/
/* Note: At this point myOutput.n holds the number of output samples generated by
* the library, where the data are written to the array pointed to by myOutput.data.
*-----------------------------------------------------------*/
```

6. Linear Transform API

This section describes the following linear transform APIs, which are supported by the DSP Library.

- Complex DFT
- Complex IDFT
- Real DFT
- Complex conjugate symmetric IDFT
- Complex FFT
- Complex IFFT
- Real FFT
- Complex conjugate symmetric IFFT

6.1 Discrete Fourier Transform (DFT)/Inverse Discrete Fourier Transform (IDFT)

6.1.1 Overview

The discrete Fourier transform (DFT) converts a time domain sequence into an equivalent-length frequency domain sequence. The frequency domain sequence is arranged in constant frequency intervals.

DFT is defined by the following formula.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j2\pi\frac{kn}{N}} \text{ , k} = 0,1,\ldots,(N-1)$$

Here, x_n is the time domain sequence samples, X_k is the frequency domain sequence samples, N is the sequence length, and $e^{-j2\pi\frac{kn}{N}}$ is the twiddle factor.

The inverse discrete Fourier transform (IDFT) converts a frequency domain sequence into an equivalent-length time domain sequence.

IDFT is defined by the following formula. The respective parameters in this formula are the same as those in the DFT formula.

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{j2\pi\frac{kn}{N}} \text{ , n} = 0,1,\ldots,(N-1)$$

6.1.2 DFT/IDFT API

The DFT/IDFT API operates input and output samples using data structures of type vector_t. Number of input samples is even number. The DFT/IDFT functions use malloc() to allocate working memory. For this reason, input and output operations can be performed as In-Place which uses the same memory.

A list of the API function formats is shown in Table 6.1, and the DFT operations on the respective combinations of input and output types are shown in Table 6.2.

Table 6.1 DFT/IDFT API Function Format List

API	Description
R_DSP_DFT_<intype><outtype>	Performs the complex DFT and real DFT operations.
R_DSP_IDFT_<intype><outtype>	Performs the complex IDFT operation.
R_DSP_IDFT_CCS_<intype><outtype>	Performs the complex conjugate symmetric IDFT operation.

Table 6.2 Correspondence Between DFT/IDFT API Function Input/Output Data Type Combinations and DFT Operations

<intype>	<outtype>	Real DFT	Complex Conjugate Symmetric IDFT	Complex DFT	Complex IDFT
i16	ci16	○	—	—	—
	ci32	○	—	—	—
i32	ci32	○	—	—	—
f32	cf32	○	—	—	—
ci16	i16	—	○	—	—
	ci16	—	—	○	○
	ci32	—	—	○	—
ci32	i16	—	○	—	—
	i32	—	○	—	—
	ci16	—	—	—	○
	ci32	—	—	○	○
cf32	f32	—	○	—	—
	cf32	—	—	○	○

The DFT/IDFT API can be divided into the complex DFT and IDFT which operate on complex numbers, and the real DFT taking real number input and complex conjugate symmetric IDFT taking as input the real DFT output constituting the first part of the frequency sequence.

- Complex DFT/complex IDFT

  The complex DFT/complex IDFT is given a complex number array as input, and a complex number array is obtained as output. The number of elements in the complex number array is even number.
- Real DFT/complex conjugate symmetric IDFT

  The real DFT takes a real number array as input, and outputs the results in a complex number array in the format shown in Figure 6.1. The DFT results from the beginning to (input sample count/2) are stored in the array.

  The complex conjugate symmetric IDFT takes a complex number array shown in Figure 6.1 as input, and outputs the IDFT results to a real number array. The number of elements of the array is twice the number of input samples.

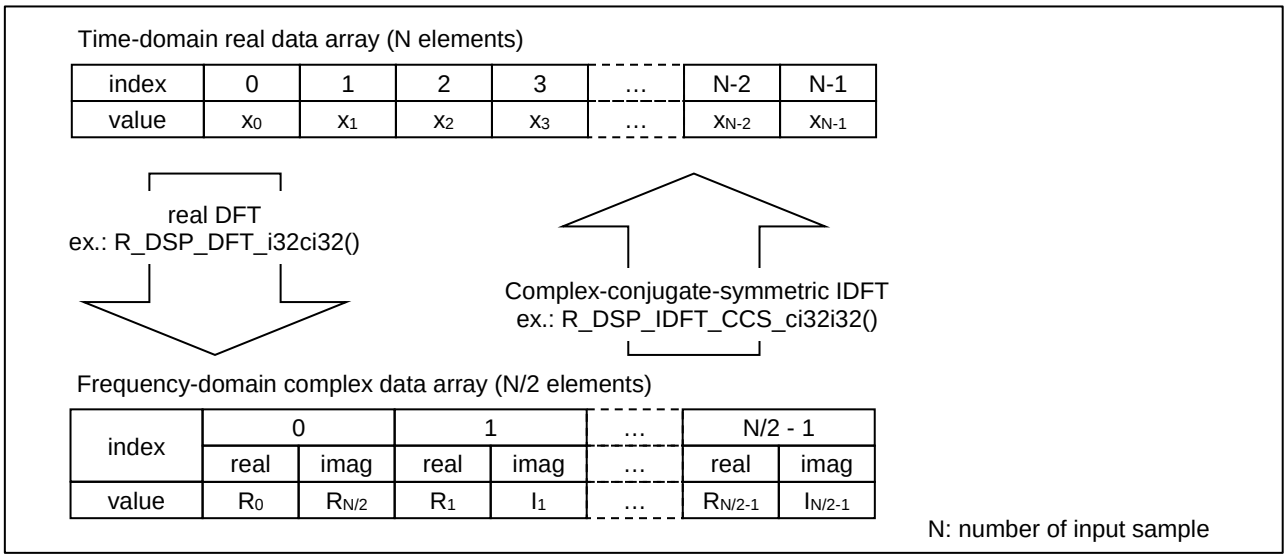


Figure 6.1 Real DFT and Complex Conjugate Symmetric IDFT Input/Output Data Storage Format

6.1.2.1 Complex DFT Functions

These functions transform a time domain complex number input array to the frequency domain sequence by DFT and store it in a complex number output array.

Format and Function List

```
r_dsp_status_t R_DSP_DFT_<intype><outtype>(
   const vector_t * src,
   vector_t * dst
)
```

Table 6.3 shows the functions for the respective combinations of <intype> and <outtype>.

Table 6.3 Complex DFT Function List

<intype>	<outtype>	Function
ci16	ci16	R_DSP_DFT_ci16ci16(const vector_t * src, vector_t * dst)
	ci32	R_DSP_DFT_ci16ci32(const vector_t * src, vector_t * dst)
ci32	ci32	R_DSP_DFT_ci32ci32(const vector_t * src, vector_t * dst)
cf32	cf32	R_DSP_DFT_cf32cf32(const vector_t * src, vector_t * dst)

Arguments

src	Pointer to a vector_t that stores the real number time domain sequence. The following members are referred to.
src->n	Input data count. The data count is even.
src->data	Pointer to the beginning of a real number array that stores the input data.
dst	Pointer to a vector_t that stores the frequency domain sequence that is output by the function. The following members are referred to.
dst->n	Element count in the array to which the data member points to. It must be src->n or greater.
dst->data	Pointer to the beginning of a complex number array that stores the calculation result.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Input vector or its data member is null.
R_DSP_ERR_OUTPUT_NULL	Output vector or its data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count data is odd.
R_DSP_ERR_INVALID_OUTPUT_SIZE	Not enough elements in output vector.
R_DSP_ERR_NO_MEMORY_AVAILABLE	malloc failed to allocate working memory.

6.1.2.2 Complex IDFT Functions

These functions transform a frequency domain complex number input array to the time domain sequence by IDFT and store it in a complex number output array.

Format and Function List

```
r_dsp_status_t R_DSP_IDFT_<intype><outtype>(
   const vector_t * src,
   vector_t * dst
)
```

Table 6.4 shows the functions for the respective combinations of <intype> and <outtype>.

Table 6.4 Complex IDFT Function List

<intype>	<outtype>	Function
ci16	ci16	R_DSP_IDFT_ci16ci16(const vector_t * src, vector_t * dst)
ci32	ci16	R_DSP_IDFT_ci32ci16(const vector_t * src, vector_t * dst)
	ci32	R_DSP_IDFT_ci32ci32(const vector_t * src, vector_t * dst)
cf32	cf32	R_DSP_IDFT_cf32cf32(const vector_t * src, vector_t * dst)

Arguments

src	Pointer to a vector_t that stores the complex number frequency domain sequence. The following members are referred to.
src->n	Input data count. The data count is even.
src->data	Pointer to the beginning of a complex number array that stores the input data.
dst	Pointer to a vector_t that stores the time domain sequence that is output by the function. The following members are referred to.
dst->n	Element count in the array. It must be src->n or greater.
dst->data	Pointer to the beginning of a complex number array that stores the calculation result.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Input vector or its data member is null.
R_DSP_ERR_OUTPUT_NULL	Output vector or its data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count data is odd.
R_DSP_ERR_INVALID_OUTPUT_SIZE	Not enough elements in output vector.
R_DSP_ERR_NO_MEMORY_AVAILABLE	malloc failed to allocate working memory.

6.1.2.3 Real DFT Functions

These functions transform a time domain real number input array to the frequency domain sequence by DFT. The real DFT results will be complex conjugate symmetric, so the first half of the frequency domain sequence is stored in a complex number output array. The input data count must be even number.

Format and Function List

```
r_dsp_status_t R_DSP_DFT_<intype><outtype>(
   const vector_t * src,
   vector_t * dst
)
```

Table 6.5 shows the functions for the respective combinations of <intype> and <outtype>.

Table 6.5 Real DFT Function List

<intype>	<outtype>	Function
i16	ci16	R_DSP_DFT_i16ci16(const vector_t * src, vector_t * dst)
	ci32	R_DSP_DFT_i16ci32(const vector_t * src, vector_t * dst)
i32	ci32	R_DSP_DFT_i32ci32(const vector_t * src, vector_t * dst)
f32	cf32	R_DSP_DFT_f32cf32(const vector_t * src, vector_t * dst)

Arguments

src	Pointer to a vector_t that stores the complex number time domain sequence. The following members are referred to.
src->n	Input data count. The data count is even.
src->data	Pointer to the beginning of a complex number array that stores the input data.
dst	Pointer to a vector_t that stores the frequency domain sequence that is output by the function. The following members are referred to.
dst->n	Element count in the array. It must be src->n/2 or greater.
dst->data	Pointer to the beginning of a complex number array that stores the calculation result.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Input vector or its data member is null.
R_DSP_ERR_OUTPUT_NULL	Output vector or its data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count data is odd.
R_DSP_ERR_INVALID_OUTPUT_SIZE	Not enough elements in output vector.
R_DSP_ERR_NO_MEMORY_AVAILABLE	malloc failed to allocate working memory.

6.1.2.4 Complex Conjugate Symmetric IDFT Functions

These functions take the first half of a complex conjugate symmetric frequency sequence as input and transform to the time domain sequence and store it in a real number output array.
The time domain real DFT results are complex conjugate symmetric, so it will be possible to reconstruct the first half of the frequency domain sequence. These functions perform the IDFT operation with the real DFT output array as input. The input data count must be even number.

Format and Function List

```
r_dsp_status_t R_DSP_IDFT_CCS_<intype><outtype>(
   const vector_t * src,
   vector_t * dst
)
```

Table 6.6 shows the functions for the respective combinations of <intype> and <outtype>.

Table 6.6 Complex Conjugate Symmetric IDFT Function List

<intype>	<outtype>	Function
ci16	i16	R_DSP_IDFT_CCS_ci16i16(const vector_t * src, vector_t * dst)
ci32	i16	R_DSP_IDFT_CCS_ci32i16(const vector_t * src, vector_t * dst)
	i32	R_DSP_IDFT_CCS_ci32i32(const vector_t * src, vector_t * dst)
cf32	f32	R_DSP_IDFT_CCS_cf32f32(const vector_t * src, vector_t * dst)

Arguments

src	Pointer to a vector_t that stores the complex number frequency domain sequence. The following members are referred to.
src->n	Input data count. The data count is even.
src->data	Pointer to the beginning of a complex number array that stores the input data.
dst	Pointer to a vector_t that stores the time domain sequence that is output by the function. The following members are referred to.
dst->n	Element count in the array to which the data member points to. It must be src->n*2 or greater.
dst->data	Pointer to the beginning of a real number array that stores the calculation result.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Input vector or its data member is null.
R_DSP_ERR_OUTPUT_NULL	Output vector or its data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is odd.
R_DSP_ERR_INVALID_OUTPUT_SIZE	Not enough elements in output vector.
R_DSP_ERR_NO_MEMORY_AVAILABLE	malloc failed to allocate working memory.

6.1.3 Example

6.1.3.1 Complex DFT/IDFT

Using the case of direct current removal in the frequency domain, an example of executing a complex DFT on 384 samples of 16-bit fixed-point complex number as input, and executing a complex IDFT on the results, is shown below. In this example, the return values from the functions are not checked.

```
#include <r_dsp_transform.h>

#define N (384)

r_dsp_status_t   my_sts = R_DSP_STATUS_OK;

cplxi16_t    x[N];  // time-domain samples. N complex samples
cplxi16_t    y[N];  // frequency-domain samples & time-domain x with DC component
removed

/* fill up x with something interesting … */
/* pointers will be cast to the correct types in the library functions */
vector_t     vin = {N, (void *)x};
vector_t     vspectrum = {N, (void *)y};

/* we define an output vector that points to the same buffer as our spectrum vector */
vector_t     vout = {N, (void *)y};

/* get the Fourier coefficients here */
my_sts = R_DSP_DFT_ci16ci16(&vin, &vspectrum);

/* knock out DC component */
y[0].re = y[0].im = 0;

/* inverse takes us back to time-domain */
/* Note that vspectrum and vout point to the same buffer in memory, */
/* therefore the function will operate in-place. */
my_sts = R_DSP_IDFT_ci16ci16 (&vspectrum, &vout);
```

6.1.3.2 Real DFT/Complex Conjugate Symmetric IDFT

Using the case of direct current removal in the frequency domain, an example of executing a real DFT on 50 of floating-point real number data as input, and executing a real IDFT on the results, is shown below. The DFT real number sequence results are complex conjugate symmetric, so the real DFT function returns the first half of the DFT results as output. This serves to save memory. In this example, the return values from the functions are not checked.

```
#include <r_dsp_transform.h>

#define      N  (50)
float        x[N];  /* time-domain samples */
cplxf32_t    y[N/2];   /* frequency-domain samples & time-domain x with DC component
removed*/

r_dsp_status_t   my_sts = R_DSP_STATUS_OK;

/* fill up x with something interesting … */
/* pointers will be cast to the correct types in the library functions */
vector_t     vin = {N, (void *)x};

/* spectrum comprises complex data, but only N/2 elements because */
/* they are complex-conjugate symmetric */
vector_t     vspectrum = {N/2, (void *)y};

/* we define an output vector that points to the same buffer as our spectrum vector */
vector_t     vout = {N, (void *)y};

/* get the Fourier coefficients here */
my_sts = R_DSP_DFT_f32cf32(&vin, &vspectrum);

/* knock out the DC component */
y[0].re = y[0].im = 0.0f;

/* inverse takes us back to time-domain */
/* Note that vspectrum and vout point to the same buffer in memory, */
/* therefore the function will operate in-place. */
my_sts = R_DSP_IDFT_CCS_cf32f32(&vspectrum, &vout);
```

6.2 Fast Fourier Transform (FFT)/Inverse Fast Fourier Transform (IFFT)

6.2.1 Overview

The fast Fourier transform (FFT) exhibits fast DFT operation by utilizing the symmetry and periodicity of a twiddle factor. The inverse fast Fourier transform (IFFT) exhibits fast DFT operation in the same manner as does the FFT. There are many FFT algorithms, but in the DSP library, a general Cooley-Tukey algorithm is implemented. It uses base 2, so the sequence length is a power of 2.

The DSP Library provides functions that are the complex FFT/IFFT which transform mutually, functions that transform input real numbers to the first half of the FFT results and functions transform input the first half of the frequency domain sequence to IFFT results as real numbers.

6.2.2 FFT/IFFT API

The FFT/IFFT API performs initialization and operation using the r_dsp_fft_t handle. The handle gives information of such as bit reversal tables, twiddle factors, windowing and operation conditions. The handle's data structure is described later.

A list of the API function formats is shown in Table 6.7, and the FFT operations on the respective combinations of input and output types are shown in Table 6.8.

Table 6.7 FFT/IFFT API Function Format List

API	Description
R_DSP_FFT_BufSize_<intype><outtype>	Returns the required size of the bit reversal table, twiddle factors and working memory.
R_DSP_FFT_Init_<intype><outtype>	Performs initialization for FFT/IFFT operation.
R_DSP_FFT_<intype><outtype>	Performs FFT operation.
R_DSP_IFFT_<intype><outtype>	Performs IFFT operation.

Table 6.8 Correspondence Between FFT/IFFT API Function Input/Output Data Type Combinations and DFT Operations

<intype>	<outtype>	Real FFT	Complex Conjugate Symmetric IFFT	Complex FFT	Complex IFFT
i16	ci16	○	—	—	—
	ci32	○	—	—	—
i32	ci32	○	—	—	—
f32	cf32	○	—	—	—
ci16	i16	—	○	—	—
	ci16	—	—	○	○
	ci32	—	—	○	—
ci32	i16	—	○	—	—
	i32	—	○	—	—
	ci16	—	—	—	○
	ci32	—	—	○	○
cf32	f32	—	○	—	—
	cf32	—	—	○	○

The FFT/IFFT API can be divided into the complex FFT and IFFT, which operate on complex numbers, and the real FFT taking real number input and complex conjugate symmetric IFFT taking as input the real FFT output constituting the first part of the frequency sequence.

- Complex FFT/Complex IFFT

  The complex FFT/complex IFFT is given a complex number array as input, and a complex number array is obtained as output. The number of elements in the complex number array is the FFT/IFFT point count.

- Real FFT/complex conjugate symmetric IFFT

  The real FFT takes as input a real number array with the size of the FFT point count and stores in a complex number output array in the way as shown in Figure 6.2. The FFT results from 0 to (point count/2) are stored in the array.

  The complex conjugate symmetric IFFT takes a complex number array in the format shown in Figure 6.2 as input, and outputs the IFFT results to a real number array. The number of elements of the array is equal to the FFT point count.

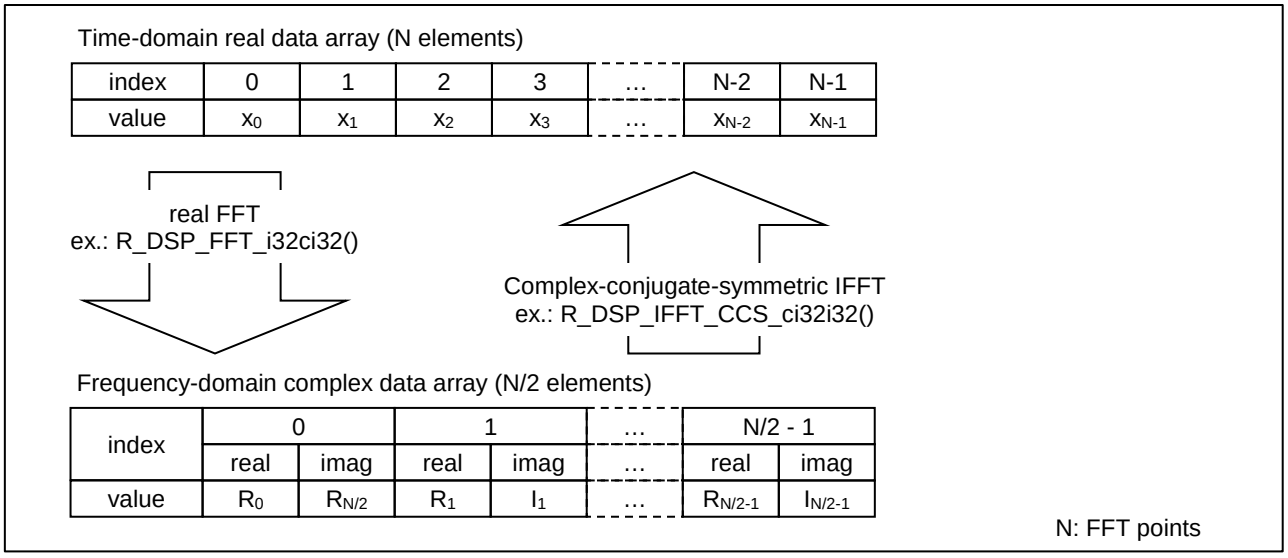


Figure 6.2 Real FFT and Complex Conjugate Symmetric IFFT Input/Output Data

The user selects <intype> and <outtype> according to the format of the input data and the format of the output data, and executes the FFT according to the procedure below.

1. Obtain the length of the twiddle factor and bit reversal arrays using R_DSP_BufSize_<intype><outtype>().
2. Define memory that is large enough to hold the twiddle factors and bit reversal table entries.
3. Execute R_DSP_FFT_Init_<intype><outtype>() to initialize the FFT handle.
4. Execute R_DSP_FFT_<intype><outtype>().

6.2.2.1 FFT Handle Data Structure

The FFT/IFFT functions use r_dsp_fft_t handle data structure. The definition of the structure is shown below.

```
typedef struct
{
   uint16_t n;               // number of points in the transform (e.g. 1024)
   uint16_t options;   // calculation options
   void * twiddles;    // twiddle factors
   void * bitrev;      // bit-reverse LUT
   void * work;        // working area
   void * window;      // window coefficients
} r_dsp_fft_t;
```

The members of the handle data structure are described in Table 6.9.

Table 6.9 Descriptions of r_dsp_fft_t Structure Members

Member	Description
n	Point count in the transform. Powers of 2 from 16 to 8192 (2^{4-13}).
options	Control options assigned to the bits. The settings below can only be specified with the fixed-point API. Option for scaling. Don't set the following two at the same time. R_DSP_FFT_OPT_SCALE: Scale the output of every FFT/IFFT stage by 1/2 R_DSP_FFT_OPT_SCALE_X2: Scale every other stage Option for twiddle factors. R_DSP_FFT_OPT_TW32: Use 32-bit twiddle factors for 16-bit fixed-point transform
twiddles	Pointer to the twiddle factor array.
bitrev	Pointer to the bit reversal lookup table.
work	Pointer to the working memory. Not available in this version of DSP library.
window	Pointer to the window coefficient array for windowing in real FFT. When windowing is not used, specify null. The data type of the coefficients is the same as the input data type. i16: Q1.15 i32: Q1.31 f32: float

6.2.2.2 FFT/IFFT Memory Size Acquisition Functions

These functions return the memory sizes of twiddle factors, bit reversal tables and working memory required for FFT/IFFT operations. The return values are shown in bytes. The respective memory sizes are calculated according to the point count n and the conditions specified in options in reference to the handle.

Format and Function List

The format of the FFT/IFFT memory size acquisition functions is as shown below.

```
r_dsp_status_t R_DSP_FFT_BufSize_<intype><outtype>(
    r_dsp_fft_t * handle,
    size_t * numTwiddleBytes,
    size_t * numBitRevBytes,
    size_t * numWorkBytes
)
```

Table 6.10 shows the functions for the respective combinations of <intype> and <outtype>.

Table 6.10 FFT/IFFT Memory Size Acquisition Function List

<intype>	<outtype>	Function	Application
i16	ci16	R_DSP_FFT_BufSize_i16ci16(r_dsp_fft_t * handle, size_t * numTwiddleBytes, size_t * numBitRevBytes, size_t * numWorkBytes)	Real FFT
	ci32	R_DSP_FFT_BufSize_i16ci32(r_dsp_fft_t * handle, size_t * numTwiddleBytes, size_t * numBitRevBytes, size_t * numWorkBytes)	
i32	ci32	R_DSP_FFT_BufSize_i32ci32(r_dsp_fft_t * handle, size_t * numTwiddleBytes, size_t * numBitRevBytes, size_t * numWorkBytes)	
f32	cf32	R_DSP_FFT_BufSize_f32cf32(r_dsp_fft_t * handle, size_t * numTwiddleBytes, size_t * numBitRevBytes, size_t * numWorkBytes)	
ci16	i16	R_DSP_FFT_BufSize_ci16i16(r_dsp_fft_t * handle, size_t * numTwiddleBytes, size_t * numBitRevBytes, size_t * numWorkBytes)	Complex conjugate symmetric IFFT
	ci16	R_DSP_FFT_BufSize_ci16ci16(r_dsp_fft_t * handle, size_t * numTwiddleBytes, size_t * numBitRevBytes, size_t * numWorkBytes)	Complex FFT, Complex IFFT
	ci32	R_DSP_FFT_BufSize_ci16ci32(r_dsp_fft_t * handle, size_t * numTwiddleBytes, size_t * numBitRevBytes, size_t * numWorkBytes)	
ci32	i16	R_DSP_FFT_BufSize_ci32i16(r_dsp_fft_t * handle, size_t * numTwiddleBytes, size_t * numBitRevBytes, size_t * numWorkBytes)	Complex conjugate symmetric IFFT
	i32	R_DSP_FFT_BufSize_ci32i32(r_dsp_fft_t * handle, size_t * numTwiddleBytes, size_t * numBitRevBytes, size_t * numWorkBytes)	
	ci16	R_DSP_FFT_BufSize_ci32ci16(r_dsp_fft_t * handle, size_t * numTwiddleBytes, size_t * numBitRevBytes, size_t * numWorkBytes)	Complex FFT, Complex IFFT
	ci32	R_DSP_FFT_BufSize_ci32ci32(r_dsp_fft_t * handle, size_t * numTwiddleBytes, size_t * numBitRevBytes, size_t * numWorkBytes)	
cf32	f32	R_DSP_FFT_BufSize_cf32f32(r_dsp_fft_t * handle, size_t * numTwiddleBytes, size_t * numBitRevBytes, size_t * numWorkBytes)	Complex conjugate symmetric IFFT
	cf32	R_DSP_FFT_BufSize_cf32cf32(r_dsp_fft_t * handle, size_t * numTwiddleBytes, size_t * numBitRevBytes, size_t * numWorkBytes)	Complex FFT, Complex IFFT

Arguments

handle	Pointer to an r_dsp_fft_t data structure. Following members are referred to. For details, see Table 6.9.
handle->n	FFT/IFFT point count.
handle->options	Option for twiddle factors. The following option can be specified. R_DSP_FFT_OPT_TW32: Use 32-bit twiddle factors for 16-bit fixed-point transform
numTwiddleBytes	Pointer to a variable storing the number of bytes required for the twiddle factors.
numBitRevBytes	Pointer to a variable storing the number of bytes required for the bit reversal table. 0 is shown when R_DSP_FFT_OPT_NO_BITREV is specified.
numWorkBytes	Pointer to a variable storing the number of bytes required for the working memory. 0 is shown when working memory is not required.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	One of the following pointers is null; numTwiddleBytes, numBitRevBytes, numWorkBytes,
R_DSP_ERR_HANDLE_NULL	Pointer to handle is null.
R_DSP_ERR_INVALID_POINTS	Point count is outside specifiable range.
R_DSP_ERR_INVALID_OPTIONS	Option specified in the handle is not supported.

6.2.2.3 FFT/IFFT Initialization Functions

These functions initialize handles to perform FFT/IFFT operations.
They refer to the handle members and create the twiddle factor array and bit reversal table based on the point count n and the conditions specified in options.
Execute one of the functions once before executing the FFT/IFFT operation functions.

Format and Function List

```
r_dsp_status_t R_DSP_FFT_Init_<intype><outtype>(
    r_dsp_fft_t * handle
)
```

Table 6.11 shows the functions for the respective combinations of <intype> and <outtype>.

Table 6.11 FFT/IFFT Initialization Function List

<intype>	<outtype>	Function	Application
i16	ci16	R_DSP_FFT_Init_i16ci16(r_dsp_fft_t * handle)	Real FFT
	ci32	R_DSP_FFT_Init_i16ci32(r_dsp_fft_t * handle)	
i32	ci32	R_DSP_FFT_Init_i32ci32(r_dsp_fft_t * handle)	
f32	cf32	R_DSP_FFT_Init_f32cf32(r_dsp_fft_t * handle)	
ci16	i16	R_DSP_FFT_Init_ci16i16(r_dsp_fft_t * handle)	Complex conjugate symmetric IFFT
	ci16	R_DSP_FFT_Init_ci16ci16(r_dsp_fft_t * handle)	Complex FFT, Complex IFFT
	ci32	R_DSP_FFT_Init_ci16ci32(r_dsp_fft_t * handle)	
ci32	i16	R_DSP_FFT_Init_ci32i16(r_dsp_fft_t * handle)	Complex conjugate symmetric IFFT
	i32	R_DSP_FFT_Init_ci32i32(r_dsp_fft_t * handle)	
	ci16	R_DSP_FFT_Init_ci32ci16(r_dsp_fft_t * handle)	Complex FFT, Complex IFFT
	ci32	R_DSP_FFT_Init_ci32ci32(r_dsp_fft_t * handle)	
cf32	f32	R_DSP_FFT_Init_cf32f32(r_dsp_fft_t * handle)	Complex conjugate symmetric IFFT
	cf32	R_DSP_FFT_Init_cf32cf32(r_dsp_fft_t * handle)	Complex FFT, Complex IFFT

Arguments

handle	Pointer to the FFT handle. Following members are referred to. For details, see Table 6.9.
handle->n	FFT/IFFT point count.
handle->options	Option for twiddle factors. The following option can be specified. R_DSP_FFT_OPT_TW32: Use 32-bit twiddle factors for 16-bit fixed-point transform
handle->twiddles	Pointer to the twiddle factor array. The array must be the value returned by R_DSP_FFT_BufSize or greater.
handle->bitrev	Pointer to the bit reversal table. The array must be the value returned by R_DSP_FFT_BufSize or greater.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_HANDLE_NULL	Pointer to handle is null.
R_DSP_ERR_INPUT_NULL	Data such as the twiddle factors or the bit reversal table is null
R_DSP_ERR_INVALID_POINTS	Point count is outside specifiable range.
R_DSP_ERR_INVALID_OPTIONS	Option specified in the handle is not supported.

6.2.2.4 Complex FFT Operation Functions

These functions perform complex FFT operations. They perform FFT on a complex number input array and store the results in an output array. With default conditions, reordering of the output is performed.
These functions do not use working memory.

Format and Function List

```
r_dsp_status_t R_DSP_FFT_<intype><outtype>(
   r_dsp_fft_t * handle,
   const vector_t * src,
   vector_t * dst
)
```

Table 6.12 shows the functions for the respective combinations of <intype> and <outtype>.

Table 6.12 Complex FFT Operation Function List

<intype>	<outtype>	Function
ci16	ci16	R_DSP_FFT_ci16ci16(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)
	ci32	R_DSP_FFT_ci16ci32(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)
ci32	ci32	R_DSP_FFT_ci32ci32(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)
cf32	cf32	R_DSP_FFT_cf32cf32(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)

Arguments

handle	Pointer to the FFT handle. The following members are referred to. For details, see Table 6.9.
handle->n	FFT point count.
handle->options	Option specification. The following option can be specified. R_DSP_FFT_OPT_TW32: Use 32-bit twiddle factors for 16-bit fixed-point transform R_DSP_FFT_OPT_SCALE: For fixed-point, scale the output of every stage by 1/2 R_DSP_FFT_OPT_SCALE_X2: For fixed-point, scale the output of every other stage by 1/2
handle->twiddles	Pointer to the twiddle factor array.
handle->bitrev	Pointer to the bit reversal table.
src	Pointer to a vector_t that stores the time domain sequence to transform. The following members are referred to.
src->n	Input data count. The count must be handle->n or greater.
src->data	Pointer to the beginning of a complex number array that stores the input data.
dst	Pointer to a vector_t that stores frequency domain sequence after FFT operation. The following members are referred to.
dst->n	Element count in the array to which the data member points to. It must be handle->n or greater.
dst->data	Pointer to the beginning of a complex number array that stores the FFT operation result.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with "_Check").
R_DSP_ERR_HANDLE_NULL	Pointer to handle is null.
R_DSP_ERR_INPUT_NULL	Input vector or data member is null, or twiddles or bitrev member of handle is null.
R_DSP_ERR_OUTPUT_NULL	Pointer to the output vector or the member data of the output vector is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Point count is outside specifiable range, or data count of the input vector is insufficient for the specified point count.
R_DSP_ERR_INVALID_OUTPUT_SIZE	Not enough elements in output vector.
R_DSP_ERR_INVALID_POINTS	Point count is outside specifiable range.
R_DSP_ERR_INVALID_OPTIONS	R_DSP_FFT_OPT_SCALE and R_DSP_FFT_OPT_SCALE_X2 are specified at the same time in the option member of the handle. This is the value returned only when using fixed-point functions.

6.2.2.5 Complex IFFT Operation Functions

These functions perform complex IFFT operations. They perform IFFT on a complex number input array and store the results in an output array. With default conditions, reordering of the output is performed.
These functions do not use working memory.

Format and Function List

```
r_dsp_status_t R_DSP_IFFT_<intype><outtype>(
   r_dsp_fft_t * handle,
   const vector_t * src,
   vector_t * dst
)
```

Table 6.13 shows the functions for the respective combinations of <intype> and <outtype>.

Table 6.13 Complex IFFT Operation Function List

<intype>	<outtype>	Function
ci16	ci16	R_DSP_IFFT_ci16ci16(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)
ci32	ci16	R_DSP_IFFT_ci32ci16(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)
	ci32	R_DSP_IFFT_ci32ci32(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)
cf32	cf32	R_DSP_IFFT_cf32cf32(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)

Arguments

handle	Pointer to the FFT handle. The following members are referred to. For details, see Table 6.9.
handle->n	IFFT point count.
handle->options	Option specification. The following option can be specified. R_DSP_FFT_OPT_TW32: Use 32-bit twiddle factors for 16-bit fixed-point transform R_DSP_FFT_OPT_SCALE: For fixed-point, scale the output of every stage by 1/2 R_DSP_FFT_OPT_SCALE_X2: For fixed-point, scale the output of every other stage by 1/2
handle->twiddles	Pointer to the twiddle factor array.
handle->bitrev	Pointer to the bit reversal table.
src	Pointer to a vector_t that stores the frequency domain sequence to transform. The following members are referred to.
src->n	Input data count. The count must be handle->n or greater.
src->data	Pointer to the beginning of a complex number array that stores the input data.
dst	Pointer to a vector_t that stores the time domain sequence after IFFT operation. The following members are referred to.
dst->n	Element count in the array to which the data member points to. It must be handle->n or greater.
dst->data	Pointer to the beginning of a complex number array that stores the calculation result.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with "_Check").
R_DSP_ERR_HANDLE_NULL	Pointer to handle is null.
R_DSP_ERR_INPUT_NULL	Input vector or data member is null, or twiddles or bitrev member of handle is null.
R_DSP_ERR_OUTPUT_NULL	Pointer to the output vector or the member data of the output vector is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Point count is outside specifiable range, or data count of the input vector is insufficient for the specified point count.
R_DSP_ERR_INVALID_OUTPUT_SIZE	Not enough elements in output vector.
R_DSP_ERR_INVALID_POINTS	Point count is outside specifiable range.
R_DSP_ERR_INVALID_OPTIONS	R_DSP_FFT_OPT_SCALE and R_DSP_FFT_OPT_SCALE_X2 are specified at the same time in the option member of the handle. This is the value returned only when using fixed-point functions.

6.2.2.6 Real FFT Operation Functions

These functions carry out real IDFT operations. They perform FFT on a real number input array, and then output complex numbers from the beginning of the results to (point count/2). With default conditions, reordering of the output is performed. These functions do not use working memory.

Format and Function List

```
r_dsp_status_t R_DSP_FFT_<intype><outtype>(
   r_dsp_fft_t * handle,
   const vector_t * src,
   vector_t * dst
)
```

Table 6.14 shows the functions for the respective combinations of <intype> and <outtype>.

Table 6.14 Real FFT Operation Function List

<intype>	<outtype>	Function
i16	ci16	R_DSP_FFT_i16ci16(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)
	ci32	R_DSP_FFT_i16ci32(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)
i32	ci32	R_DSP_FFT_i32ci32(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)
f32	cf32	R_DSP_FFT_f32cf32(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)

Arguments

handle	Pointer to the FFT handle. The following members are referred to. For details, see Table 6.9.
handle->n	FFT point count.
handle->options	Option specification. The following option can be specified. R_DSP_FFT_OPT_TW32: Use 32-bit twiddle factors for 16-bit fixed-point transform R_DSP_FFT_OPT_SCALE: For fixed-point, scale the output of every stage by 1/2 R_DSP_FFT_OPT_SCALE_X2: For fixed-point, scale the output of every other stage by 1/2
handle->twiddles	Pointer to the twiddle factor array.
handle->bitrev	Pointer to the bit reversal table.
handle->windows	Pointer to the window coefficient array for windowing. When windowing is not used, specify null. The data type of coefficients must be the same as input data type.
src	Pointer to a vector_t that stores the time domain sequence to transform. The following members are referred to.
src->n	Input data count. It must be handle->n or greater.
src->data	Pointer to the beginning of a complex number array that stores the input data.
dst	Pointer to a vector_t that stores frequency domain sequence after FFT operation. The following members are referred to.
dst->n	Element count in the array to which the data member points to. It must be (handle->n)/2 or greater.
dst->data	Pointer to the beginning of a complex number array that stores the FFT operation result. The storage order of the input data is as follows. (N: FFT point count)

index	0		1		…	N/2-1	
	Real	Imag	Real	Imag	…	Real	Imag
value	R_0	$R_{N/2}$	R_1	I_1	…	$R_{N/2-1}$	$I_{N/2-1}$

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with "_Check").
R_DSP_ERR_HANDLE_NULL	Pointer to handle is null.
R_DSP_ERR_INPUT_NULL	Point count is outside specifiable range, or data count of the input vector is insufficient for the specified point count.
R_DSP_ERR_OUTPUT_NULL	Pointer to the output vector or the member data of the output vector is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Points is outside specifiable range, or data count of the input vector is insufficient for the specified point count.
R_DSP_ERR_INVALID_OUTPUT_SIZE	Not enough elements in output vector.
R_DSP_ERR_INVALID_POINTS	Point count is outside specifiable range.
R_DSP_ERR_INVALID_OPTIONS	R_DSP_FFT_OPT_SCALE and R_DSP_FFT_OPT_SCALE_X2 are specified at the same time in the option member of the handle. This is the value returned only when using fixed-point functions.

6.2.2.7 Complex Conjugate Symmetric IFFT Operation Functions

These functions perform IFFT operations. They perform IFFT on an array of complex numbers in a frequency sequence from the beginning of the sequence to (point count/2), and then output real numbers with the output count equal to the point count. With default conditions, reordering of the output is performed.
These functions do not use working memory.
The twiddle factors for a complex conjugate FFT differ from those of a real FFT. The user must generate a twiddle factor array for these functions by calling the corresponding FFT initialization function.

Format and Function List

```
r_dsp_status_t R_DSP_IFFT_CCS_<intype><outtype>(
   r_dsp_fft_t * handle,
   const vector_t * src,
   vector_t * dst
)
```

Table 6.15 shows the functions for the respective combinations of <intype> and <outtype>.

Table 6.15 Complex Conjugate Symmetric IFFT Function List

<intype>	<outtype>	Function
ci16	i16	R_DSP_IFFT_CCS_ci16i16(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)
ci32	i16	R_DSP_IFFT_CCS_ci32i16(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)
	i32	R_DSP_IFFT_CCS_ci32i32(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)
cf32	f32	R_DSP_IFFT_CCS_cf32f32(r_dsp_fft_t * handle, const vector_t * src, vector_t * dst)

Arguments

<table>
<tr><td>handle</td><td>Pointer to the FFT handle. The following members are referred to. For details, see Table 6.9.</td></tr>
<tr><td>handle->n</td><td>IFFT point count.</td></tr>
<tr><td>handle->options</td><td>Option specification. The following option can be specified.
R_DSP_FFT_OPT_TW32: Use 32-bit twiddle factors for 16-bit fixed-point transform
R_DSP_FFT_OPT_SCALE: For fixed-point, scale the output of every stage by 1/2
R_DSP_FFT_OPT_SCALE_X2: For fixed-point, scale the output of every other stage by 1/2</td></tr>
<tr><td>handle->twiddles</td><td>Pointer to the twiddle factor array.</td></tr>
<tr><td>handle->bitrev</td><td>Pointer to the bit reversal table.</td></tr>
<tr><td>src</td><td>Pointer to a vector_t that stores the frequency domain sequence to transform. The following members are referred to.</td></tr>
<tr><td>src->n</td><td>Input data count. It must be (handle->n)/2 or greater.</td></tr>
<tr><td>src->data</td><td>Pointer to the beginning of a complex number array that stores the input data. The storage order of the input data is as follows. (N: IFFT point count)
<table>
<tr><td rowspan="2">Index</td><td colspan="2">0</td><td colspan="2">1</td><td>…</td><td colspan="2">N/2-1</td></tr>
<tr><td>Real</td><td>Imag</td><td>Real</td><td>Imag</td><td>…</td><td>Real</td><td>Imag</td></tr>
<tr><td>Value</td><td>R_0</td><td>$R_{N/2}$</td><td>R_1</td><td>I_1</td><td>…</td><td>$R_{N/2-1}$</td><td>$I_{N/2-1}$</td></tr>
</table></td></tr>
<tr><td>dst</td><td>Pointer to a vector_t that stores the time domain sequence after IFFT operation. The following members are referred to.</td></tr>
<tr><td>dst->n</td><td>Element count in the array to which the data member points to. It must be handle->n or greater.</td></tr>
<tr><td>dst->data</td><td>Pointer to the beginning of a real number array that stores the IFFT operation result.</td></tr>
</table>

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with "_Check").
R_DSP_ERR_HANDLE_NULL	Pointer to handle is null.
R_DSP_ERR_INPUT_NULL	Input vector or data member is null, or twiddles or bitrev member of handle is null.
R_DSP_ERR_OUTPUT_NULL	Pointer to the output vector or the member data of the output vector is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Point count is outside specifiable range, or data count of the input vector is insufficient for the specified point count.
R_DSP_ERR_INVALID_OUTPUT_SIZE	Not enough elements in output vector.
R_DSP_ERR_INVALID_POINTS	Point count is outside specifiable range.
R_DSP_ERR_INVALID_OPTIONS	R_DSP_FFT_OPT_SCALE and R_DSP_FFT_OPT_SCALE_X2 are specified at the same time in the option member of the handle. This is the value returned only when using fixed-point functions.

6.2.3 Example

6.2.3.1 Complex FFT

A sample program that performs a 256-point complex FFT (16-bit input/output, 32-bit twiddle factor, output reordered) is shown below. The operation is performed in place, with the input array and output array using the same memory. In this example, the return values from the functions are not checked.

```
#include <r_dsp_transform.h>

#define FFT_POINTS (256)

cplxi16_t      fft_twiddles[FFT_POINTS]; // twiddle table for 256 points complex FFT
uint32_t       fft_bitrev[120];   // bit reverse table for 256 points complex FFT
r_dsp_fft_t    h = {FFT_POINTS, R_DSP_FFT_OPT_TW32, fft_twiddles. fft_bitrev, NULL, NULL};
r_dsp_status_t my_sts = R_DSP_STATUS_OK;

/* same data buffer x is referenced for both input and output, */
cplxi16_t      x[FFT_POINTS]; // time-domain complex samples (interleaved i and q)
vector_t       vin = {FFT_POINTS, (void *)x};
vector_t       vout = {FFT_POINTS, (void *)x};


/* initialize required buffer sizes for twiddles etc. */
size_t         ntwb; // bytes for twiddle array
size_t         nbrb; // bytes for bit-reverse table
size_t         nwkb; // bytes for working area

/* get the required size given the FFT size, input/output data types, etc */
my_sts = R_DSP_FFT_BufSize_ci16ci16(&h, &ntwb, &nbrb, &nwkb);
if((sizeof(fft_twiddles) < ntwb) || (sizeof(fft_bitrev) < ntrb))
{
    while(1);  /* not enough memory size */
}
/* in the complex FFT case, nwkb is always 0, so no need to set data */
/* in the complex FFT case, h.window is not used ,so no need to set data */

/* initialize twiddle table and bit-reversal table */
/* check status for any error, e.g. NULL pointers */
my_sts = R_DSP_FFT_Init_ci16ci16(&h);

/* With the handle now fully initialized, we now invoke the forward FFT function: */
/* the FFT function will operate in-place */
my_sts = R_DSP_FFT_ci16ci16(&h, &vin, &vout);

/* x now holds the Fourier coefficients */
```

6.2.3.2 Complex FFT/IFFT

A sample program that performs a 32-point complex FFT/IFFT (floating-point input/output, output reordered) is shown below. In this example, the return values from the functions are not checked.

```
#include <r_dsp_transform.h>

#define FFT_POINTS (32)

/* initialize handle structure */
cplxf32_t      fft_twiddles[FFT_POINTS]; // twiddle table for 32 points complex FFT
uint32_t       fft_bitrev[12];    // bit reverse table for 32 points complex FFT
r_dsp_fft_t    h = {FFT_POINTS, 0, fft_twiddles. fft_bitrev, NULL, NULL}; /* default option
set*/
r_dsp_status_t my_sts = R_DSP_STATUS_OK;

/* With the handle initialized, we may now invoke the forward FFT function: */
cplxf32_t  buf_time[FFT_POINTS]; /* time-domain complex samples */
cplxf32_t  buf_freq[FFT_POINTS]; /* frequency-domain complex samples */
vector_t   vtime = {FFT_POINTS , (void *)buf_time};
vector_t   vfreq = {FFT_POINTS , (void *)buf_freq};

/* initialize required buffer sizes for twiddles etc. */
size_t     ntwb;   // bytes for twiddle array
size_t     nbrb;   // bytes for bit-reverse table
size_t     nwkb;   // bytes for working area

/* get the required size given the FFT size, input/output data types, etc. */
my_sts = R_DSP_FFT_BufSize_cf32cf32(&h, &ntwb, &nbrb, &nwkb);
if((sizeof(fft_twiddles) < ntwb) || (sizeof(fft_bitrev) < ntrb))
{
    while(1);  /* not enough memory size */
}
/* in the inverse complex FFT case, nwkb is always 0, so no needs to set data */
/* in the inverse complex FFT case, h.window is not used, so no needs to set data */

/* now we can initialize the handle */
/* check status for any error, e.g. NULL pointers */
my_sts = R_DSP_FFT_Init_cf32cf32(&h);

/* With the handle initialized, we may now invoke the forward FFT function:*/
my_sts = R_DSP_FFT_cf32cf32(&h, &vtime, &vfreq);

/* perform the inverse FFT and then de-allocate any resources: */
my_sts = R_DSP_IFFT_cf32cf32(&h, &vfreq, &vtime);
```

6.2.3.3 Real FFT/Complex Conjugate Symmetric IFFT

A sample program that performs a 32-point real FFT and then, using that output as input, performs a complex conjugate symmetric IFFT (floating-point input/output, output reordered) is shown below. In this example, the return values from the functions are not checked.

```
#include <r_dsp_transform.h>

#define FFT_POINTS (32)

int32_t    windowCoefficients[FFT_POINTS]; /* coefficients for windowing in Q1.31 format */
cplxi32_t  fft_twiddles[FFT_POINTS + FFT_POINTS/2]; // twiddle table for 32 points complex
FFT
uint32_t   fft_bitrev[12];    // bit reverse table for 32 points complex FFT
/* options: default
 * window: if windowing is necessary; otherwise set NULL
 */
r_dsp_fft_t    h = {FFT_POINTS, 0, fft_twiddles. fft_bitrev, NULL, windowCoefficients};
r_dsp_status_t my_sts = R_DSP_STATUS_OK;

/* initialize handle structure */

size_t     ntwb;      // bytes for twiddle array
size_t     nbrb;      // bytes for bit-reverse table
size_t     nwkb;      // bytes for working area

int32_t    buf_time[FFT_POINTS];     // real-valued time-domain samples */
cplxi32_t  buf_freq[FFT_POINTS/2];   // complex-valued frequency-domain data */
vector_t   vtime = {FFT_POINTS, (void *)buf_time};
vector_t   vfreq = {FFT_POINTS/2, (void *)buf_freq};

/* initialize required buffer sizes for twiddles etc. */
/* NOTE real-valued input specified! */
my_sts = R_DSP_FFT_BufSize_i32ci32(&h, &ntwb, &nbrb, &nwkb);
if((sizeof(fft_twiddles) < ntwb) || (sizeof(fft_bitrev) < ntrb))
{
    while(1);  /* not enough memory size */
}
/* in the real FFT case, nwkb is always 0, so no needs to set data */

/* now we can complete initialization of the handle */
/* NOTE real-valued input specified in the function name */
my_sts = R_DSP_FFT_Init_i32ci32(&h);

/* With the handle initialized, we may now invoke the forward FFT function:*/
/* only get back the unique portion of the complex-conjugate-symmetric FFT */
my_sts = R_DSP_FFT_i32ci32(&h, &vtime, &vfreq);

/* perform the inverse FFT with the handle re-initialized */
my_sts = R_DSP_FFT_Init_ci32i32(&h);
my_sts = R_DSP_IFFT_CCS_ci32i32(&h, &vfreq, &vtime);
```

7. Complex Number Operation API

This section describes the following complex number operation functions, which are supported by the DSP Library.

- Complex number magnitude functions
- Complex number magnitude squared functions
- Complex number phase functions
- Complex number addition functions
- Complex number subtraction functions
- Complex number multiplication functions
- Complex conjugate functions

7.1 Complex Number Magnitude Functions

These functions calculate the magnitude of scalar complex number and vectors of complex numbers. The scalar and vector versions of the functions both in turn have two versions: a precise type, and a fast type that avoids time-consuming square root operations. The names of the fast versions of the functions contain the string "_Fast".

7.1.1 Scalar Version

The scalar version functions return the magnitude of a scalar complex number. With the 16-bit fixed-point data type, the precision level is cut in half as a result of square roots, so functions that return 32-bit results are provided in order to preserve precision.

Format and Function List

```
<outtype2> R_DSP_CplxMag_<intype1><outtype1>(<intype2> x)
<outtype2> R_DSP_CplxMag_Fast_<intype1><outtype1>(<intype2> x)
```

<intype1> and <intype2> are the same data type. Likewise, <outtype1> and <outtype2> are the same data type. Table 7.1 shows the functions for the respective combinations of <intype> and <outtype>.

Table 7.1 Complex Number Magnitude Function (Scalar Version) List

<intype1> <intype2>	<outtype1> <outtype2>	Function (Upper: Precise Type; Lower: Fast Type)
ci16 cplxi16_t	i16 int16_t	int16_t R_DSP_CplxMag_ci16i16(cplxi16_t x) int16_t R_DSP_CplxMag_Fast_ci16i16(cplxi16_t x)
	i32 int32_t	int32_t R_DSP_CplxMag_ci16i32(cplxi16_t x) int32_t R_DSP_CplxMag_Fast_ci16i32(cplxi16_t x)
ci32 cplxi32_t	i32 int32_t	int32_t R_DSP_CplxMag_ci32i32(cplxi32_t x) int32_t R_DSP_CplxMag_Fast_ci32i32(cplxi32_t x)
cf32 cplxf32	f32 float	float R_DSP_CplxMag_cf32f32(cplxf32_t x) float R_DSP_CplxMag_Fast_cf32f32(cplxf32_t x)

Arguments

x	A complex number.

Return Values

The magnitude of the complex number ranges [0, 2]. Then in case of a fix-point number, return values are Q2.14 format when outputs are specified as i16 and Q2.30 format when outputs are specified as i32.

Description

The precise-type functions without "_Fast" perform their calculations according to the formula below. In the formula, a represents the real part of the complex number, and b represents the imaginary part of the complex number.

$$mag = \sqrt{a^2 + b^2}$$

The fast-type functions with "_Fast" use the linear least-squares method in the formula below to avoid calculating square roots. As a result, there is an error incidence of approximately 4% in the results.

$$mag = 0.961 \cdot \max(|a|, |b|) + 0.397 \cdot min(|a|, |b|)$$

7.1.2 Vector Version

The vector version functions calculate the magnitude of each complex element in a vector and store the results in an output vector. With the 16-bit fixed-point data type, the precision level is cut in half as a result of square roots, so functions that return 32-bit results are provided to preserve precision.

Format and Function List

The functions are shown below. The upper is the precision type and the lower is the fast type.

```
r_dsp_status_t R_DSP_VecCplxMag_<intype><outtype>(
  const vector_t * input,
  vector_t * output,
  const uint32_t N
)

r_dsp_status_t R_DSP_VecCplxMag_Fast_<intype><outtype>(
  const vector_t * input,
  vector_t * output,
  const uint32_t N
)
```

Table 7.2 shows the functions for the respective combinations of <intype> and <outtype>.

Table 7.2 Complex Number Magnitude Function (Vector Version) List

<intype>	<outtype>	Function (Upper: Precise Type; Lower: Fast Type)
ci16	i16	R_DSP_VecCplxMag_ci16i16(const vector_t * input, vector_t * output, const unsigned int N) R_DSP_VecCplxMag_Fast_ci16i16(const vector_t * input, vector_t * output, const unsigned int N)
	i32	R_DSP_VecCplxMag_ci16i32(const vector_t * input, vector_t * output, const unsigned int N) R_DSP_VecCplxMag_Fast_ci16i32(const vector_t * input, vector_t * output, const unsigned int N)
ci32	i32	R_DSP_VecCplxMag_ci32i32(const vector_t * input, vector_t * output, const unsigned int N) R_DSP_VecCplxMag_Fast_ci32i32(const vector_t * input, vector_t * output, const unsigned int N)
cf32	f32	R_DSP_VecCplxMag_cf32f32(const vector_t * input, vector_t * output, const unsigned int N) R_DSP_VecCplxMag_Fast_cf32f32(const vector_t * input, vector_t * output, const unsigned int N)

Arguments

input	Pointer to a vector_t that stores the input data. The following members are referred to.
input->n	Input data count.
input->data	Pointer to the beginning of a complex number array that stores the input data.
output	Pointer to a vector that stores the magnitudes that are output by the function. The following members are referred to.
output ->n	Number of elements in the array to which the data member points to. Specify N (described below) or greater.
output ->data	Pointer to the beginning of a real number array.
N	Used in cases in which, based on the option parameter specifying the data count on which to operate, only a portion of the input data is operated on. Specify a value not exceeding input->n. Specify 0 if the input data count for the operation is input->n.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Pointer to input vector or its data member is null.
R_DSP_ERR_OUTPUT_NULL	Pointer to output vector or its data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	The value of N is non-zero and input->n or greater. Or input->n is 0.
R_DSP_ERR_INVALID_OUTPUT_SIZE	output->n is too small for output data count.

Description

See 7.1.1, Scalar Version.

Example

An example of the use of a complex magnitude function with real 16-bit fixed-point input and 32-bit fixed-point output is shown below. The same method is used for the other data formats as well.

```
#include <r_dsp_complex.h>
#define N (384)

cplxi16_t x[N];  /* complex samples */
int32_t   mag[N];   /*|x|*/
r_dsp_status_t my_sts = R_DSP_STATUS_OK; // place to store return status

/* fill up x with something interesting … */
vector_t  vx = {N, x};
vector_t  vmag = {N, mag};

/* accurate computation of compute magnitude here */
my_sts = R_DSP_VecCplxMag_ci16i32(&vx, &vmag, N);

/* a faster computation */
my_sts = R_DSP_VecCplxMag_Fast_ci16i32(&vx, &vmag, N);
```

7.2 Complex Number Magnitude Squared Functions

These functions calculate the square of the magnitude of scalar complex numbers and vectors of complex numbers.

7.2.1 Scalar Version

The scalar function versions return the square of the magnitude of a scalar complex number. With the 16-bit fixed-point data type, functions that return 32-bit results are provided in order to preserve precision.

Format and Function List

```
<outtype2> R_DSP_CplxMagSquared_<intype1><outtype1>(<intype2> x)
```

<intype1> and <intype2> are the same data type. Likewise, <outtype1> and <outtype2> are the same data type. Table 7.3 shows the functions for the respective combinations of <intype> and <outtype>.

Table 7.3 Complex Number Magnitude Squared Function (Scalar Version) List

<intype1> <intype2>	<outtype1> <outtype2>	Function
ci16 cplxi16_t	i16 int16_t	int16_t R_DSP_CplxMagSquared_ci16i16(cplxi16_t x)
	i32 int32_t	int32_t R_DSP_CplxMagSquared_ci16i32(cplxi16_t x)
ci32 cplxi32_t	i32 int32_t	int32_t R_DSP_CplxMagSquared_ci32i32(cplxi32_t x)
cf32 cplxf32	f32 float	float R_DSP_CplxMagSquared_cf32f32(cplxf32_t x)

Arguments

x	A complex number.

Return Values

The magnitude of the complex number ranges [0,2]. Then, in case of fix-point number, return values are Q3.13 format when outputs are specified as i16 and Q3.29 format when outputs are specified as i32.

Description

The square of the magnitude of a complex number is calculated according to the following formula.

$$mag^2 = a^2 + b^2$$

In the formula, a represents the real part of the complex number, and b represents the imaginary part of the complex number.

7.2.2 Vector Version

The vector function versions calculate the square of the magnitude of each element in a vector and store the results in an output vector. With the 16-bit fixed-point data type, functions that return 32-bit results are provided to preserve precision.

Format and Function List

```
r_dsp_status_t R_DSP_VecCplxMagSquared_<intype><outtype>(
   const vector_t * input,
   vector_t * output,
   const uint32_t N
)
```

Table 7.4 shows the functions for the respective combinations of <intype> and <outtype>.

Table 7.4 Complex Number Magnitude Squared Function (Vector Version) List

<intype>	<outtype>	Function (Upper: Precise Type; Lower: Fast Type)
ci16	i16	R_DSP_VecCplxMagSquared_ci16i16(const vector_t * input, vector_t * output, const unsigned int N)
	i32	R_DSP_VecCplxMagSquared_ci16i32(const vector_t * input, vector_t * output, const unsigned int N)
ci32	i32	R_DSP_VecCplxMagSquared_ci32i32(const vector_t * input, vector_t * output, const unsigned int N)
cf32	f32	R_DSP_VecCplxMagSquared_cf32f32(const vector_t * input, vector_t * output, const unsigned int N)

Arguments

input	Pointer to a vector_t that stores the input data. The following members are referred to.
input->n	Input data count.
input->data	Pointer to the beginning of a complex number array that stores the input data.
output	Pointer to a vector that stores the magnitudes that are output by the function. The following members are referred to.
output ->n	Number of elements in the array to which the data member points to. Specify N (described below) or greater.
output ->data	Pointer to the beginning of a real number array.
N	Option parameter specifying the data count to operate. This option is used when only a portion of the input data is operated on. Specify a value not exceeding input->n. Specify 0 when the input data count for the operation is input->n.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Pointer to input vector or its data member is null.
R_DSP_ERR_OUTPUT_NULL	Pointer to output vector or its data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	The value of N is non-zero and input->n or greater.
R_DSP_ERR_INVALID_OUTPUT_SIZE	output->n is too small for output data count.

Description

See 7.2.1, Scalar Version.

Example

An example of the use of a complex magnitude function with real 16-bit fixed-point input and 32-bit fixed-point output is shown below. The same method is used for the other data formats as well.

```
#include <r_dsp_complex.h>
#define N (384)

cplxi16_t x[N]; /* complex samples */
int32_t   mag2[N]; /* |x|^2 */
r_dsp_status_t my_sts = R_DSP_STATUS_OK; // place to store return status

/* fill up x with something interesting … */
vector_t  vx = {N, x};
vector_t  vmag2 = {N, mag2};

/* accurate computation of compute magnitude squared here */
my_sts = R_DSP_VecCplxMagSquared_ci16i32(&vx, &vmag2, N);
```

7.3 Complex Number Phase Functions

These functions calculate the phase of scalar complex numbers and vectors of complex numbers.

7.3.1 Scalar Version

The scalar version functions return the phase of a scalar complex number.

Format and Function List

```
<outtype2> R_DSP_CplxPhase_<intype1><outtype1>(<intype2> x)
```

<intype1> and <intype2> are the same data type. Likewise, <outtype1> and <outtype2> are the same data type. Table 7.5 shows the functions for the respective combinations of <intype> and <outtype>.

Table 7.5 Complex Number Phase Function List (Scalar Version)

<intype1> <intype2>	<outtype1> <outtype2>	Function
ci16 cplxi16_t	i16 int16_t	int16_t R_DSP_CplxPhase_ci16i16(cplxi16_t x)
ci32 cplxi32_t	i32 int32_t	int32_t R_DSP_CplxPhase_ci32i32(cplxi32_t x)
cf32 cplxf32	f32 float	float R_DSP_CplxPhase_cf32f32(cplxf32_t x)

Arguments

x	A complex number.

Return Values

The phase of the complex number ranges [-π, π]. Then in case of a fix-point number, return values are Q3.13 format when outputs are specified as i16 and Q3.29 format when outputs are specified as i32.

Description

The phase of complex number z is calculated according to the following formula. In the formula, x represents the real part of the complex number, and y represents the imaginary part of the complex number.

$$\varphi = \arg(z) = \begin{cases} arctan\left(\frac{y}{x}\right), & x > 0 \\ arctan\left(\frac{y}{x}\right) + \pi, & x < 0 \ and \ y \geq 0 \\ arctan\left(\frac{y}{x}\right) - \pi, & x < 0 \ and \ y < 0 \\ \frac{\pi}{2}, & x = 0 \ and \ y > 0 \\ -\frac{\pi}{2}, & x = 0 \ and \ y < 0 \\ 0, & x = 0 \ and \ y = 0 \end{cases}$$

If the real and imaginary parts of the complex number are both 0, it will not be possible to determine the phase, so these functions will return 0.

7.3.2 Vector Version

The vector version functions calculate the phase of each element in a vector and store the results in an output vector.

Format and Function List

```
r_dsp_status_t R_DSP_VecCplxPhase_<intype><outtype>(
   const vector_t * input,
   vector_t * output,
   const unsigned int N
)
```

Table 7.6 shows the functions for the respective combinations of <intype> and <outtype>.

Table 7.6 Complex Number Phase Function List (Vector Version)

<intype>	<outtype>	Function (Upper: Precise Type; Lower: Fast Type)
ci16	i16	R_DSP_VecCplxPhase_ci16i16(const vector_t * input, vector_t * output, const unsigned int N)
ci32	i32	R_DSP_VecCplxPhase_ci32i32(const vector_t * input, vector_t * output, const unsigned int N)
cf32	f32	R_DSP_VecCplxPhase_cf32f32(const vector_t * input, vector_t * output, const unsigned int N)

Arguments

input	Pointer to a vector_t that stores the input data. The following members are referred to.
input->n	Input data count.
input->data	Pointer to the beginning of a complex number array that stores the input data.
output	Pointer to a vector that stores the phase value that are output by the function. The values range [-π, π]. Then in case of fix-point number, the values are Q3.13 format when outputs specified as i16 and Q3.29 format when outputs specified as i32. The following members are referred to.
output ->n	Number of elements in the array to which the data member points to. Specify N (described below) or greater.
output ->data	Pointer to the beginning of a real number array.
N	Option parameter specifying the data count to operate. This option is used when only a portion of the input data is operated on. Specify a value not exceeding input->n. Specify 0 when the input data count for the operation is input->n.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_UNDEFINED_RESULT	Real and imaginary parts of input element complex number are both 0.
R_DSP_ERR_INPUT_NULL	Pointer to input or data member is null.
R_DSP_ERR_OUTPUT_NULL	Pointer to output or data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	input element count is 0 or smaller than N.
R_DSP_ERR_INVALID_OUTPUT_SIZE	output element count is 0 or smaller than N.

Description

See 7.3.1 Scalar Version. If the real and imaginary parts of the complex number are both 0, it will not be possible to determine the phase, so these functions will return R_DSP_STATUS_UNDEFINED_RESULT.

Example

An example of the use of a complex number phase function on a 16-bit fixed-point complex number is shown below. The same method is used for the other data formats as well.

```
#include <r_dsp_complex.h>
#define N (384)

cplxi16_t x[N]; /* complex samples */
int16_t   phase[N]; /* phase of x */
r_dsp_status_t my_sts = R_DSP_STATUS_OK; // place to store return status

/* fill up x with something interesting … */
vector_t  vx = {N, x};
vector_t  vphase = {N, phase};

my_sts = R_DSP_VecCplxPhase_ci16i16(&vx, &vphase, N);
```

7.4 Complex Number Addition Functions

These functions add two scalar complex numbers.
In case of the fixed-point functions, formats of input and output are same.

Format and Function List

```
r_dsp_status_t R_DSP_ComplexAdd_<intype1><outtype1>(
   const <intype2> inputA,
   const <intype2> inputB,
   <outtype2> * sum
)
```

<intype1> and <outtype1> are the same data type. Specify <intype2> to be cplxi16, cplxi32, or cplxf32 in accordance with <intype1>, and specify <outtype2> to be cplxi16, cplxi32, or cplxf32 in accordance with <outtype1>. Table 7.7 shows the functions for the respective combinations of <intype1>, <intype2>, <outtype1>, and <outtype2>.

Table 7.7 Complex Number Addition Function List

<intype1> <intype2>	<outtype1> <outtype2>	Function
ci16 cplxi16_t	ci16 cplxi16_t	R_DSP_ComplexAdd_ci16ci16(const cplxi16_t inputA, const cplxi16_t inputB, cplxi16_t * sum)
ci32 cplxi32_t	ci32 cplxi32_t	R_DSP_ ComplexAdd_ci32ci32(const cplxi32_t inputA, const cplxi32_t inputB, cplxi32_t * sum)
cf32 cplxf32_t	cf32 cplxf32_t	R_DSP_ ComplexAdd_cf32cf32(const cplxf32_t inputA, const cplxf32_t inputB, cplxf32_t * sum)

Arguments

inputA	One of the complex numbers to be added.
inputB	The other complex number to be added.
sum	Pointer to a complex number that stores the operation result.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with "_Check").
R_DSP_ERR_OUTPUT_NULL	sum is null.

Description

Taking the input complex numbers A and B, and the output complex number C, the complex number addition is performed according to the formula below.

$$C = A + B = B + A = \begin{cases} real(C) = real(A) + real(B) \\ imag(C) = imag(A) + imag(B) \end{cases}$$

In the formula above, $real()$ and $imag()$ represent the real and imaginary parts of the complex numbers.

Example

An example of the use of a 16-bit fixed-point complex number addition function is shown below.

```
#include <r_dsp_complex.h>

cplxi16_t inputA, inputB, sum;
r_dsp_status_t my_sts = R_DSP_STATUS_OK; // place to store return status

my_sts = R_DSP_ComplexAdd_ci16ci16(inputA, inputB, &sum);
```

7.5 Complex Number Subtraction Functions

These functions subtract one scalar complex number from another.
In case of the fixed-point functions, formats of input and output are same.

Format and Function List

```
r_dsp_status_t R_DSP_ComplexSub_<intype1><outtype1>(
   const <intype2> inputA,
   const <intype2> inputB,
   <outtype2> * diff
)
```

<intype1> and <outtype1> are the same data type. Specify <intype2> to be cplxi16, cplxi32, or cplxf32 in accordance with <intype1>, and specify <outtype2> to be cplxi16, cplxi32, or cplxf32 in accordance with <outtype1>. Table 7.8 shows the functions for the respective combinations of <intype1>, <intype2>, <outtype1>, and <outtype2>.

Table 7.8 Complex Number Subtraction Function List

<intype1> <intype2>	<outtype1> <outtype2>	Function
ci16 cplxi16_t	ci16 cplxi16_t	R_DSP_ComplexSub_ci16ci16(const cplxi16_t inputA, const cplxi16_t inputB, cplxi16_t * diff)
ci32 cplxi32_t	ci32 cplxi32_t	R_DSP_ ComplexSub_ci32ci32(const cplxi32_t inputA, const cplxi32_t inputB, cplxi32_t * diff)
cf32 cplxf32_t	cf32 cplxf32_t	R_DSP_ ComplexSub_cf32cf32(const cplxf32_t inputA, const cplxf32_t inputB, cplxf32_t * diff)

Arguments

inputA	The complex number to be subtracted from.
inputB	The complex number to be subtracted.
diff	Pointer to a complex number that stores the operation result.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with "_Check").
R_DSP_ERR_OUTPUT_NULL	diff is null.

Description

Taking the input complex numbers to be A and B, and the output complex number to be C, the complex number subtraction is performed according to the formula below.

$$C = A - B = \begin{cases} real(C) = real(A) - real(B) \\ imag(C) = imag(A) - imag(B) \end{cases}$$

In the formula above, $real()$ and $imag()$ represent the real and imaginary parts of the complex numbers.

Example

An example of the use of a 16-bit fixed-point complex number subtraction function is shown below.

```
#include <r_dsp_complex.h>

cplxi16_t inputA, inputB, diff;
r_dsp_status_t my_sts = R_DSP_STATUS_OK; // place to store return status

my_sts = R_DSP_ComplexSub_ci16ci16(inputA, inputB, &diff);
```

7.6 Complex Number Multiplication Functions

These functions multiply two scalar complex numbers.
For fixed-point operation functions, the output format is Q1.xx. In the example where the input format is Q1.xx, the output format is Q1.15 for ci16ci16 multiplication, the output format is Q1.31 for ci16ci32 and ci32ci32 multiplication.

Format and Function List

```
r_dsp_status_t R_DSP_ComplexMul_<intype1><outtype1>(
   const <intype2> inputA,
   const <intype2> inputB,
   <outtype2> * product
)
```

<intype1> and <outtype1> are the same data type. Specify <intype2> to be cplxi16, cplxi32, or cplxf32 in accordance with <intype1>, and specify <outtype2> to be cplxi16, cplxi32, or cplxf32 in accordance with <outtype1>. Table 7.9 shows the functions for the respective combinations of <intype1>, <intype2>, <outtype1>, and <outtype2>.

Table 7.9 Complex Number Multiplication Function List

<intype1> <intype2>	<outtype1> <outtype2>	Function
ci16 cplxi16_t	ci16 cplxi16_t	R_DSP_ComplexMul_ci16ci16(const cplxi16_t inputA, const cplxi16_t inputB, cplxi16_t * product)
	ci32 cplxi32_t	R_DSP_ComplexMul_ci16ci32(const cplxi16_t inputA, const cplxi16_t inputB, cplxi32_t * product)
ci32 cplxi32_t	ci32 cplxi32_t	R_DSP_ ComplexMul_ci32ci32(const cplxi32_t inputA, const cplxi32_t inputB, cplxi32_t * product)
cf32 cplxf32_t	cf32 cplxf32_t	R_DSP_ ComplexMul_cf32cf32(const cplxf32_t inputA, const cplxf32_t inputB, cplxf32_t * product)

Arguments

inputA	One of the complex numbers to be multiplied.
inputB	The other complex number to be multiplied.
product	Pointer to a complex number that stores the operation result.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with "_Check").
R_DSP_ERR_OUTPUT_NULL	Pointer to variable that stores the result is null.

Description

Taking the input complex numbers A and B, and the output complex number C, the complex number multiplication is performed according to the formula below.

$$\mathrm{C} = \mathrm{A} \cdot \mathrm{B} = \mathrm{B} \cdot \mathrm{A} = \begin{cases} real(C) = real(A) \cdot real(B) - imag(A) \cdot imag(B) \\ imag(C) = real(A) \cdot imag(B) + imag(A) \cdot real(B) \end{cases}$$

Above, *real()* and *imag()* represent the real and imaginary parts of the complex numbers.

Example

An example of the use of a 16-bit fixed-point complex number multiplication function is shown below.

```
#include <r_dsp_complex.h>

cplxi16_t inputA, inputB, product;
r_dsp_status_t my_sts = R_DSP_STATUS_OK; // place to store return status

my_sts = R_DSP_ComplexMul_ci16ci16(inputA, inputB, &product);
```

7.7 Complex Conjugate Functions

These functions calculate the complex conjugate of scalar complex numbers and vectors of complex numbers.
In case of the fixed-point functions, the input and output formats are the same.

7.7.1 Scalar Version

The scalar version functions output the complex conjugate of an input scalar complex number.

Format and Function List

```
r_dsp_status_t R_DSP_CplxConjg_<intype1><outtype1>(
   const <intype2> input,
   <outtype2> * output
)
```

<intype1> and <outtype1> are the same data type. Specify <intype2> to be cplxi16, cplxi32, or cplxf32 in accordance with <intype1>, and specify <outtype2> to be cplxi16, cplxi32, or cplxf32 in accordance with <outtype1>. Table 7.10 shows the functions for the respective combinations of <intype1>, <intype2>, <outtype1>, and <outtype2>.

Table 7.10 Complex Conjugate Function List (Scalar Version)

<intype1> <intype2>	<outtype1> <outtype2>	Function
ci16 cplxi16_t	ci16 cplxi16_t	R_DSP_CplxConjg_ci16ci16(const cplxi16_t input cplxi16_t * output)
ci32 cplxi32_t	ci32 cplxi32_t	R_DSP_CplxConjg_ci32ci32(const cplxi32_t input cplxi32_t * output)
cf32 cplxf32_t	cf32 cplxf32_t	R_DSP_CplxConjg_cf32cf32(const cplxf32_t input cplxf32_t * output)

Arguments

input	A complex number.
output	Pointer to a complex number that stores the operation result.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with “_Check”).
R_DSP_ERR_OUTPUT_NULL	output is null.

Description

Taking the input complex number A, and the output complex number B, the complex conjugate is calculated according to the following formula.

$$B = A^* = \begin{cases} real(B) = real(A) \\ imag(B) = -imag(A) \end{cases}$$

In the formula above, *real()* and *imag()* represent the real and imaginary parts of the complex numbers.

7.7.2 Vector Version

The vector version functions calculate the complex conjugate of each element in a vector and store the results in an output vector.

Format and Function List

```
r_dsp_status_t R_DSP_VecCplxConjg_<intype><outtype>(
   const vector_t * input,
   vector_t * output
)
```

Table 7.11 shows the functions for the respective combinations of <intype> and <outtype>.

Table 7.11 Complex Conjugate Function List (Vector Version)

<intype>	<outtype>	Function
ci16	ci16	R_DSP_VecCplxConjg_ci16ci16(const vector_t * input vector_t * output)
	ci32	R_DSP_VecCplxConjg_ci16ci32(const vector_t * input vector_t * output)
ci32	ci32	R_DSP_VecCplxConjg_ci32ci32(const vector_t * input vector_t * output)
cf32	cf32	R_DSP_VecCplxConjg_cf32f32(const vector_t * input vector_t * output)

Arguments

input	Pointer to a vector_t that stores the input data. The following members are referred to...
input->n	Input data count.
input->data	Pointer to the beginning of a complex number array that stores the input data.
output	Pointer to a vector_t that stores the output of the function. The following members are referred to..
output ->n	Number of elements in the array to which the data member points to.
output ->data	Pointer to the beginning of a complex number array.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with “_Check”).
R_DSP_ERR_INPUT_NULL	Pointer to input vector or its data member is null.
R_DSP_ERR_OUTPUT_NULL	Pointer to output vector or its data member is null.
R_DSP_ERR_INVALID_INPUT_SIZE	Input data count is 0
R_DSP_ERR_INVALID_OUTPUT_SIZE	output->n is too small for output data count.

Description

See 7.7.1, Scalar Version.

Example

An example of the use of a 32-bit fixed-point complex conjugate function is shown below.

```
#include <r_dsp_complex.h>

vector_t  input, output;
r_dsp_status_t my_sts = R_DSP_STATUS_OK; // place to store return status

my_sts = R_DSP_VecCplxConjg_ci32ci32((const vector_t *)&input, &output);
```

8. Matrix Operation API

This section describes the following matrix operation functions, which are supported by the DSP library.

- Matrix addition functions
- Matrix subtraction functions
- Matrix multiplication functions
- Matrix transposition functions
- Matrix real number multiplication functions

8.1 Matrix Addition Functions

These functions add two input matrices A and B. They add each element in input matrix A to the corresponding element in input matrix B. The result of the addition is stored in the corresponding element of an output matrix. The generated output matrix has the same dimensions as A or B.

Format and Function List

```
r_dsp_status_t R_DSP_MatrixAdd_<intype><outtype>(
   const matrix_t * inputA,
   const matrix_t * inputB,
   matrix_t * output
)
```

Table 8.1 shows the functions for the respective combinations of <intype> and <outtype>.

Table 8.1 Matrix Addition Function List

<intype>	<outtype>	Function
i16	i16	R_DSP_MatrixAdd_i16i16(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)
	i32	R_DSP_MatrixAdd_i16i32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)
ci16	i16	R_DSP_MatrixAdd_ci16ci16(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)
	i32	R_DSP_MatrixAdd_ci16ci32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)
i32	i32	R_DSP_MatrixAdd_i32i32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)
ci32	ci32	R_DSP_MatrixAdd_ci32ci32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)
f32	f32	R_DSP_MatrixAdd_f32f32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)
cf32	cf32	R_DSP_MatrixAdd_cf32cf32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)

Arguments

inputA	Pointer to one of the two matrices to be added. The matrix structure and the data to which the pointers in the structure point are not modified by these functions. The following members are referred to.
inputA->nRows	Number of rows in the matrices.
inputA->nCols	Number of columns in the matrices.
inputA->data	Pointer to the first element of a matrix.
inputB	Pointer to the other of the two matrices to be added. The matrix structure and the data to which the pointers in the structure point are not modified by these functions. The following members are referred to.
inputB->nRows	Number of rows in the matrices.
inputB->nCols	Number of columns in the matrices.
inputB->data	Pointer to the first element of a matrix.
output	Pointer to the output matrix to store operation results. The following members are referred to.
output ->nRows	Number of rows in the matrices. The function updates this.
output ->nCols	Number of columns in the matrices. The function updates this.
output ->data	Pointer to the first element of a matrix.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with "_Check").
R_DSP_ERR_INPUT_NULL	Pointer to input matrix or data member is null.
R_DSP_ERR_OUTPUT_NULL	Pointer to output matrix or data member is null.
R_DSP_ERR_DIMENSIONS	The matrix dimensions are invalid or mismatched.

Description

Taking the input matrices to be *A* and *B*, and the output matrix to be *C*, the matrix addition is executed according to the operation below.

$$\mathrm{C} = \begin{bmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,M-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,M-1} \\ \vdots & \vdots & \vdots & \vdots \\ c_{N-1,0} & c_{N-1,1} & \cdots & c_{N-1,M-1} \end{bmatrix} = A + B = B + \mathrm{A}$$

$$= \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,M-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,M-1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{N-1,0} & a_{N-1,1} & \cdots & a_{N-1,M-1} \end{bmatrix} + \begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,M-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,M-1} \\ \vdots & \vdots & \vdots & \vdots \\ b_{N-1,0} & b_{N-1,1} & \cdots & b_{N-1,M-1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0} + b_{0,0} & a_{0,1} + \mathrm{b}_{0,1} & \cdots & a_{0,M-1} + \mathrm{b}_{0,M-1} \\ a_{1,0} + b_{1,0} & a_{1,1} + b_{1,1} & \cdots & a_{1,M-1} + \mathrm{b}_{0,M-1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{N-1,0} + b_{N-1,0} & a_{N-1,1} + b_{N-1,1} & \cdots & a_{N-1,M-1} + \mathrm{b}_{N-1,M-1} \end{bmatrix}$$

Above, *N* is the number of rows, and *M* is the number of columns.

Example

An example of the use of a real 32-bit fixed-point input/output matrix addition function is shown below. The same method is used for the other data formats as well.

```
#include <r_dsp_matrix.h>
#define NUM_ROWS      (4)
#define NUM_COLUMNS   (3)
int32_t   dataLeft[NUM_ROWS][NUM_COLUMNS];
int32_t   dataRight[NUM_ROWS][NUM_COLUMNS];
int32_t   dataOut[NUM_ROWS][NUM_COLUMNS];
matrix_t  channLeft = {NUM_ROWS, NUM_COLUMNS, (void *)dataLeft};
matrix_t  channRight = {NUM_ROWS, NUM_COLUMNS, (void *)dataRight};
matrix_t  channOut;
r_dsp_status_t my_sts = R_DSP_STATUS_OK; // place to store return status

/* Set up the matrices */
channOut.data = (void *) dataOut;

/* Call the matrix add function. */
/* Note the depending on the data type variation, void pointers will be cast appropriately
within the function.*/
my_sts = R_DSP_MatrixAdd_i32i32 (&channLeft, &channRight, &channOut);
```

8.2 Matrix Subtraction Functions

These functions subtract each element in input matrix B from the corresponding element in input matrix A. The result of subtraction is stored in the corresponding element of an output matrix. The generated output matrix has the same dimensions as A or B.

Format and Function List

```
r_dsp_status_t R_DSP_MatrixSub_<intype><outtype>(
   const matrix_t * inputA,
   const matrix_t * inputB,
   matrix_t * output
)
```

Table 8.2 shows the functions for the respective combinations of <intype> and <outtype>.

Table 8.2 Matrix Subtraction Function List

<intype>	<outtype>	Function
i16	i16	R_DSP_MatrixSub_i16i16(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)
	i32	R_DSP_MatrixSub_i16i32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)
ci16	i16	R_DSP_MatrixSub_ci16ci16(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)
	i32	R_DSP_MatrixSub_ci16ci32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)
i32	i32	R_DSP_MatrixSub_i32i32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)
ci32	ci32	R_DSP_MatrixSub_ci32ci32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)
f32	f32	R_DSP_MatrixSub_f32f32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)
cf32	cf32	R_DSP_MatrixSub_cf32cf32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output)

Arguments

inputA	Pointer to the matrix to be subtracted from. The matrix structure and the data to which the pointers in the structure point are not modified by these functions. The following members are referred to.
inputA->nRows	Number of rows in the matrices.
inputA->nCols	Number of columns in the matrices.
inputA->data	Pointer to the first element of a matrix.
inputB	Pointer to the matrix to subtract. The matrix structure and the data to which the pointers in the structure point are not modified by these functions. The following members are referred to.
inputB->nRows	Number of rows in the matrices.
inputB->nCols	Number of columns in the matrices.
inputB->data	Pointer to the first element of a matrix.
output	Pointer to the output matrix to store operation results. The following members are referred to.
output ->nRows	Number of rows in the matrices. The function updates this.
output ->nCols	Number of columns in the matrices. The function updates this.
output ->data	Pointer to the first element of a matrix.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with "_Check").
R_DSP_ERR_INPUT_NULL	Pointer to input matrix or data member is null.
R_DSP_ERR_OUTPUT_NULL	Pointer to output matrix or data member is null.
R_DSP_ERR_DIMENSIONS	The matrix dimensions are invalid or mismatched.

Description

Taking the input matrices to be A and B, and the output matrix to be C, the matrix subtraction is executed according to the operation below.

$$\mathrm{C} = \begin{bmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,M-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,M-1} \\ \vdots & \vdots & \vdots & \vdots \\ c_{N-1,0} & c_{N-1,1} & \cdots & c_{N-1,M-1} \end{bmatrix} = A - B$$

$$= \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,M-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,M-1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{N-1,0} & a_{N-1,1} & \cdots & a_{N-1,M-1} \end{bmatrix} - \begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,M-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,M-1} \\ \vdots & \vdots & \vdots & \vdots \\ b_{N-1,0} & b_{N-1,1} & \cdots & b_{N-1,M-1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0} - b_{0,0} & a_{0,1} - \mathrm{b}_{0,1} & \cdots & a_{0,M-1} - \mathrm{b}_{0,M-1} \\ a_{1,0} - b_{1,0} & a_{1,1} - b_{1,1} & \cdots & a_{1,M-1} - \mathrm{b}_{0,M-1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{N-1,0} - b_{N-1,0} & a_{N-1,1} - b_{N-1,1} & \cdots & a_{N-1,M-1} - \mathrm{b}_{N-1,M-1} \end{bmatrix}$$

Above, N is the number of rows, and M is the number of columns.

Example

An example of the use of a real 32-bit fixed-point input/output matrix subtraction function is shown below. The same method is used for the other data formats as well.

```
#include <r_dsp_matrix.h>
#define NUM_ROWS      (4)
#define NUM_COLUMNS   (3)
int32_t   dataLeft[NUM_ROWS][NUM_COLUMNS];
int32_t   dataRight[NUM_ROWS][NUM_COLUMNS];
int32_t   dataOut[NUM_ROWS][NUM_COLUMNS];
matrix_t  channLeft = {NUM_ROWS, NUM_COLUMNS, (void *)dataLeft};
matrix_t  channRight = {NUM_ROWS, NUM_COLUMNS, (void *)dataRight};
matrix_t  channOut;
r_dsp_status_t my_sts = R_DSP_STATUS_OK; // place to store return status

/* Set up the matrices */
channOut.data = (void *) dataOut;

/* Call the matrix subtract function. */
/* Note the depending on the data type variation, void pointers will be cast appropriately
within the function. */
my_sts = R_DSP_MatrixSub_i32i32 (&channLeft, &channRight, &channOut);
```

8.3 Matrix Multiplication Functions

These functions calculate a matrix product by multiplying two matrices. The input matrix supplied as the first argument is the multiplicand, and the input matrix supplied as the second argument is the multiplier. The element in the *i*th row and *j*th column of the matrix product is the dot product of the *i*th row of the multiplicand and the *j*th column of the multiplier.
In general, matrix multiplication is not commutative, and the multiplier and multiplicand cannot be switched. If they are switched, either the operation will become invalid or a different matrix product will result.
The number of columns in the multiplicand matrix is the same as the number of rows in the multiplier matrix. The output matrix will have the same number of rows as the multiplicand and the same number of columns as the multiplier.

Format and Function List

```
r_dsp_status_t R_DSP_MatrixMul_<intype><outtype>(
   const matrix_t * inputA,
   const matrix_t * inputB,
   matrix_t * output,
   scale_t shift,
   uint16_t options
)
```

Table 8.3 shows the functions for the respective combinations of <intype> and <outtype>.

Table 8.3 Matrix Multiplication Function List

<intype>	<outtype>	Function
i16	i16	R_DSP_MatrixMul_i16i16(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output, scale_t shift, uint16_t options)
	i32	R_DSP_MatrixMul_i16i32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output, scale_t shift, uint16_t options)
ci16	ci16	R_DSP_MatrixMul_ci16ci16(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output, scale_t shift, uint16_t options)
	ci32	R_DSP_MatrixMul_ci16ci32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output, scale_t shift, uint16_t options)
i32	i32	R_DSP_MatrixMul_i32i32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output, scale_t shift, uint16_t options)
ci32	ci32	R_DSP_MatrixMul_ci32ci32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output, scale_t shift, uint16_t options)
f32	f32	R_DSP_MatrixMul_f32f32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output, scale_t shift, uint16_t options)
cf32	cf32	R_DSP_MatrixMul_cf32cf32(const matrix_t * inputA, const matrix_t * inputB, matrix_t * output, scale_t shift, uint16_t options)

Arguments

inputA	Pointer to the multiplicand matrix. The matrix structure and the data to which the pointers in the structure point are not modified by these functions. The following members are referred to.
inputA->nRows	Number of rows in the matrices.
inputA->nCols	Number of columns in the matrices.
inputA->data	Pointer to the first element of a matrix.
inputB	Pointer to the multiplier matrix. The matrix structure and the data to which the pointers in the structure point are not modified by these functions. The following members are referred to.
inputB->nRows	Number of rows in the matrices.
inputB->nCols	Number of columns in the matrices.
inputB->data	Pointer to the first element of a matrix.
output	Pointer to the output matrix where operation results are stored. The following members are referred to.
output ->nRows	Number of rows in the matrices. The function updates this.
output ->nCols	Number of columns in the matrices. The function updates this.
output ->data	Pointer to the first element of a matrix.
shift	Output data scaling parameter. For details, see 2.9 Scaling. For fixed-point operations, the operation result is right-shifted corresponding to this value. The scaling parameter is an integer, and the valid value ranges are as follows. i32i32, ci32ci32 format: 1 to 62 i16i16, ci16ci16 format: 1 to 30 i16i32, ci16ci32 format: -31 to +31 (negative values indicate left-shifting) For floating-point operations, the operation results are multiplied by this value. The scaling parameter is a floating-point value. When the value is greater than 1.0, the results are amplified. When the value is smaller than 1.0, the results are attenuated.
options	Operation control options. The fixed-point operation functions support rounding modes and saturation modes. The floating-point operation functions do not refer to them. The following two saturation modes are supported. R_DSP_NOSATURATE (default) R_DSP_SATURATE The following two rounding modes are supported. R_DSP_ROUNDING_TRUNC (default) R_DSP_ROUNDING_NEAREST

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with "_Check").
R_DSP_ERR_INPUT_NULL	Pointer to input matrix or data member is null.
R_DSP_ERR_OUTPUT_NULL	Pointer to output matrix or data member is null.
R_DSP_ERR_DIMENSIONS	The matrix dimensions are invalid or mismatched.
R_DSP_ERR_INVALID_SCALE	Value of shift member of handle is outside specifiable range.

Description

Taking the multiplicand matrix A, the multiplier to be B, and the matrix product C, the matrix multiplication is executed according to the operation below.

$$\mathrm{C} = \begin{bmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,K-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,K-1} \\ \vdots & \vdots & \vdots & \vdots \\ c_{N-1,0} & c_{N-1,1} & \cdots & c_{N-1,K-1} \end{bmatrix} = A \times B$$

$$= \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,M-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,M-1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{N-1,0} & a_{N-1,1} & \cdots & a_{N-1,M-1} \end{bmatrix} \times \begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,K-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,K-1} \\ \vdots & \vdots & \vdots & \vdots \\ b_{M-1,0} & b_{M-1,1} & \cdots & b_{M-1,K-1} \end{bmatrix}$$

$$= \begin{bmatrix} \sum_{j=0}^{M-1} a_{0,j} \cdot b_{j,0} & \sum_{j=0}^{M-1} a_{0,j} \cdot b_{j,1} & \cdots & \sum_{j=0}^{M-1} a_{0,j} \cdot b_{j,K-1} \\ \sum_{j=0}^{M-1} a_{1,j} \cdot b_{j,0} & \sum_{j=0}^{M-1} a_{1,j} \cdot b_{j,1} & \cdots & \sum_{j=0}^{M-1} a_{1,j} \cdot b_{j,K-1} \\ \vdots & \vdots & \vdots & \vdots \\ \sum_{j=0}^{M-1} a_{N-1,j} \cdot b_{j,0} & \sum_{j=0}^{M-1} a_{N-1,j} \cdot b_{j,1} & \cdots & \sum_{j=0}^{M-1} a_{N-1,j} \cdot b_{j,K-1} \end{bmatrix}$$

Above, N is the number of rows in A, M is the number of columns in A and the number of rows in B, and K is the number of columns in B. The product of the matrices will have N rows and K columns.

Example

An example of the use of a real 32-bit fixed-point input/output matrix multiplication function is shown below. The same method is used for the other data formats as well.

```
#include <r_dsp_matrix.h>

#define NUM_ROWS_A    (4)
#define NUM_COLUMNS_A (3)
#define NUM_ROWS_B    NUM_COLUMNS_A
#define NUM_COLUMNS_B (4)

int32_t   dataLeft[NUM_ROWS_A][NUM_COLUMNS_A];
int32_t   dataRight[NUM_ROWS_B][NUM_COLUMNS_B];
int32_t   dataOut[NUM_ROWS_A][NUM_COLUMNS_B];
matrix_t  channLeft = {NUM_ROWS_A, NUM_COLUMNS_A, (void *)dataLeft};
matrix_t  channRight = {NUM_ROWS_B, NUM_COLUMNS_B, (void *)dataRight};
matrix_t  channOut;
scale_t   shift;
r_dsp_status_t my_sts = R_DSP_STATUS_OK; // place to store return status

/* Set up the matrices */
channOut.data = (void *) dataOut;
shift.i32 = 31; // scaling factor for the output data to get result as Q1.31

/* Call the matrix multiply function.
/* Note the depending on the data type variation, void pointers will be cast appropriately
within the function.*/
my_sts = R_DSP_MatrixMul_i32i32 (&channLeft, &channRight, &channOut, shift, NULL);
```

8.4 Matrix Transposition Functions

These functions transpose an input matrix. An output matrix is generated by switching the columns and rows of the input matrix. Element j in the *i*th row of the output matrix will be element i in the *j*th row of the input matrix.
The number of rows in the output matrix will be the same as the number of columns in the input matrix, and the number of columns in the output matrix will be the same as the number of rows in the input matrix.

Format and Function List

```
r_dsp_status_t R_DSP_MatrixTrans_<intype><outtype>(
   const matrix_t * input,
   matrix_t * output,
)
```

<intype> and <outtype> must be the same type. Table 8.4 shows the functions for the respective combinations of <intype> and <outtype>.

Table 8.4 Matrix Transposition Function List

<intype>	<outtype>	Function
i16	i16	R_DSP_MatrixTrans_i16i16(const matrix_t * input, matrix_t * output)
	i32	R_DSP_MatrixTrans_i16i32(const matrix_t * input, matrix_t * output)
ci16	i16	R_DSP_MatrixTrans_ci16ci16(const matrix_t * inputA, matrix_t * output)
	ci32	R_DSP_MatrixTrans_ci16ci32(const matrix_t * input, matrix_t * output)
i32	i32	R_DSP_MatrixTrans_i32i32(const matrix_t * inputA, matrix_t * output)
ci32	ci32	R_DSP_MatrixTrans_ci32ci32(const matrix_t * inputA, matrix_t * output)
f32	f32	R_DSP_MatrixTrans_f32f32(const matrix_t * inputA, matrix_t * output)
cf32	cf32	R_DSP_MatrixTrans_cf32cf32(const matrix_t * inputA, matrix_t * output)

Arguments

input	Pointer to the input matrix. The matrix structure and the data to which the pointers in the structure point are not modified by these functions. The following members are referred to.
input->nRows	Number of rows in the matrices.
input->nCols	Number of columns in the matrices.
input->data	Pointer to the first element of a matrix.
output	Pointer to the output matrix. The following members are referred to.
output ->nRows	Number of rows in the matrices. The function updates this.
output ->nCols	Number of columns in the matrices. The function updates this.
output ->data	Pointer to the first element of a matrix.

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_ERR_INPUT_NULL	Pointer to input matrix or data member is null.
R_DSP_ERR_OUTPUT_NULL	Pointer to output matrix or data member is null.
R_DSP_ERR_DIMENSIONS	The matrix dimensions are invalid or mismatched.

Description

Taking the input matrix A and the output matrix B, the matrix transposition functions are executed according to the operation below.

$$\mathrm{B} = A^{T} = transpose\left(\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,M-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,M-1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{N-1,0} & a_{N-1,1} & \cdots & a_{N-1,M-1} \end{bmatrix}\right) = \begin{bmatrix} a_{0,0} & a_{1,0} & \cdots & a_{\mathrm{N}-1,0} \\ a_{0,1} & a_{1,1} & \cdots & a_{\mathrm{N}-1,1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{0,\mathrm{M}-1} & a_{1,\mathrm{M}-1} & \cdots & a_{N-1,M-1} \end{bmatrix}$$

Above, N and M respectively represent the number of rows and number of columns in the input matrix, and the number of rows and number of columns in the output matrix as well.

Example

An example of the use of a real 32-bit fixed-point input/output matrix transposition function is shown below. The same method is used for the other data formats as well.

```
#include <r_dsp_matrix.h>
#define NUM_ROWS      (4)
#define NUM_COLUMNS   (3)
int32_t   dataInput[NUM_ROWS][NUM_COLUMNS];
int32_t   dataOutput[NUM_COLUMNS][NUM_ROWS];
matrix_t  channInput = {NUM_ROWS, NUM_COLUMNS, (void *)dataInput};
matrix_t  channOutput;
r_dsp_status_t my_sts = R_DSP_STATUS_OK; // place to store return status

/* Set up the matrices */
channOutput.data = (void *) dataOutput;

/* Call the matrix transpose function. */
/* Note the depending on the data type variation, void pointers will be cast appropriately
within the function. */
my_sts = R_DSP_MatrixTrans_i32i32(&channInput, &channOutput);
```

8.5 Matrix Real Number Multiplication Functions

These functions multiply an input matrix by a scalar value. Each element of the output matrix is generated by scalar multiplication of the corresponding element of the input matrix. The dimensions of the output matrix will be the same as those of the input matrix.
The scalar and the input data must be the same data type.

Format and Function List

```
r_dsp_status_t R_DSP_MatrixScale_<intype1><outtype>(
   const matrix_t * input,
   const <intype2> scalar,
   matrix_t * output,
   scale_t shift,
   uint16_t options
)
```

Specify <intype2> to be the same type as <intype1>: int16_t, int32_t, or float. Table 8.5 shows the functions for the respective combinations of <intype1>, <intype2>, and <outtype>.

Table 8.5 Matrix Real Number Multiplication Function List

<intype1>	<intype2>	<outtype>	Function
i16	int16_t	i16	R_DSP_MatrixScale_i16i16(const matrix_t * input, const int16_t scalar, matrix_t * output, scale_t shift, uint16_t options)
		i32	R_DSP_MatrixScale_i16i32(const matrix_t * input, const int16_t scalar, matrix_t * output, scale_t shift, uint16_t options)
ci16	int16_t	ci16	R_DSP_MatrixScale_ci16ci16(const matrix_t * input, const int16_t scalar, matrix_t * output, scale_t shift, uint16_t options)
		ci32	R_DSP_MatrixScale_ci16ci32(const matrix_t * input, const int16_t scalar, matrix_t * output, scale_t shift, uint16_t options)
i32	int32_t	i32	R_DSP_MatrixScale_i32i32(const matrix_t * input, const int32_t scalar, matrix_t * output, scale_t shift, uint16_t options)
ci32	int32_t	ci32	R_DSP_MatrixScale_ci32ci32(const matrix_t * input, const int32_t scalar, matrix_t * output, scale_t shift, uint16_t options)
f32	float	f32	R_DSP_MatrixScale_f32f32(const matrix_t * input, const float_t scalar, matrix_t * output, scale_t shift, uint16_t options)
cf32	float	cf32	R_DSP_MatrixScale_cf32cf32(const matrix_t * input, const float scalar, matrix_t * output, scale_t shift, uint16_t options)

Arguments

input	Pointer to the input matrix. The matrix structure and the data to which the pointers in the structure point are not modified by these functions. The following members are referred to.
input->nRows	Number of rows in the matrices.
input->nCols	Number of columns in the matrices.
input->data	Pointer to the first element of a matrix.
scalar	The value by which to multiply each element of the matrix. It must be the same data type as the input data.
output	Pointer to the output matrix where operation results are stored. The following members are referred to.
output ->nRows	Number of rows in the matrices. The function updates this.
output ->nCols	Number of columns in the matrices. The function updates this.
output ->data	Pointer to the first element of a matrix.
shift	Output data scaling parameter. For details, see 2.9 Scaling. For fixed-point operations, the operation result is right-shifted corresponding to this value. The scaling parameter is an integer, and the valid value ranges are as follows. i32i32, ci32ci32 format: 1 to 62 i16i16, ci16ci16 format: 1 to 30 i16i32, ci16ci32 format: -31 to +31 (negative values indicate left-shifting) For floating-point operations, the operation results are multiplied by this value. The scaling parameter is a floating-point value. When the value is greater than 1.0, the results are amplified. When the value is smaller than 1.0, the results are attenuated.
options	Operation control options. The fixed-point operation functions support rounding modes and saturation modes. The floating-point operation functions do not refer to them. The following two saturation modes are supported. R_DSP_NOSATURATE (default) R_DSP_SATURATE The following two rounding modes are supported. R_DSP_ROUNDING_TRUNC (default) R_DSP_ROUNDING_NEAREST

Return Values

R_DSP_STATUS_OK	Normal exit.
R_DSP_STATUS_OVERFLOW	Overflow occurrence (in case of fixed-point functions of the library with "_Check").
R_DSP_ERR_INPUT_NULL	Pointer to input matrix or data member is null.
R_DSP_ERR_OUTPUT_NULL	Pointer to output matrix or data member is null.
R_DSP_ERR_DIMENSIONS	The matrix dimensions are invalid or mismatched.
R_DSP_ERR_INVALID_SCALE	Value of shift member of handle is outside specifiable range.

Description

Taking the input matrices A and the output matrix B, the real number multiplication functions are executed according to the operation below.

$$\mathrm{B} = \mathrm{A} \cdot \mathrm{k} = k \cdot A = k \cdot \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,M-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,M-1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{N-1,0} & a_{N-1,1} & \cdots & a_{N-1,M-1} \end{bmatrix} = \begin{bmatrix} k \cdot a_{0,0} & k \cdot a_{1,0} & \cdots & k \cdot a_{\mathrm{N}-1,0} \\ k \cdot a_{0,1} & k \cdot a_{1,1} & \cdots & k \cdot a_{\mathrm{N}-1,1} \\ \vdots & \vdots & \vdots & \vdots \\ k \cdot a_{0,\mathrm{M}-1} & k \cdot a_{1,\mathrm{M}-1} & \cdots & k \cdot a_{N-1,M-1} \end{bmatrix}$$

Above, N and M respectively represent the number of rows and number of columns in the input matrix, and k represents the scalar value.

Example

An example of the use of a real 32-bit fixed-point input/output matrix real number multiplication function is shown below. The same method is used for the other data formats as well.

```
#include <r_dsp_matrix.h>
#define NUM_ROWS      (4)
#define NUM_COLUMNS   (3)

int32_t   dataInput[NUM_ROWS][NUM_COLUMNS];
int32_t   dataOutput[NUM_ROWS][NUM_COLUMNS];
matrix_t  channInput = {NUM_ROWS, NUM_COLUMNS, (void *)dataInput};
matrix_t  channOutput;
int32_t   gainFactor = 2;
scale_t   shift;
r_dsp_status_t my_sts = R_DSP_STATUS_OK; // place to store return status

/* Set up the matrices */
channOutput.data = (void *) dataOutput;
shift.i32 = 1; // scaling factor for the output data as Q2.30

/* Call the matrix scale function. */
/* Note the depending on the data type variation, void pointers will be cast appropriately
within the function. */
my_sts = R_DSP_MatrixScale_i32i32 (&channInput, gainFactor, &channOutput, shift, NULL);
```

Revision History	RX DSP Library APIs Version 5.0 User's Manual: Software

Rev.	Date	Description	
		Page	Summary
1.00	Jan 21, 2019	—	First Edition issued

RX DSP Library APIs Version 5.0 User’s Manual: Software

Publication Date: Rev.1.00 Jan 21, 2019

Published by: Renesas Electronics Corporation

SALES OFFICES

Renesas Electronics Corporation

http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics Corporation
TOYOSU FORESIA, 3-2-24 Toyosu, Koto-ku, Tokyo 135-0061, Japan

Renesas Electronics America Inc.
1001 Murphy Ranch Road, Milpitas, CA 95035, U.S.A.
Tel: +1-408-432-8888, Fax: +1-408-434-5351

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-651-700

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709 Quantum Plaza, No.27 ZhichunLu, Haidian District, Beijing, 100191 P. R. China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, 200333 P. R. China
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HAL 2nd Stage, Indiranagar, Bangalore 560 038, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
17F, KAMCO Yangjae Tower, 262, Gangnam-daero, Gangnam-gu, Seoul, 06265 Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5338

Colophon 5.2

RX DSP Library APIs Version 5.0
User's Manual: Software

RX Family

Renesas Electronics Corporation

R01UW0200EJ0100