To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

RENESAS

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of 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. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. 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 of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may 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.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   “Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   “High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   “Specific”: Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, 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, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be 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, or if you have any other inquiries.

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

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

SuperH RISC engine C/C++ Compiler Package

APPLICATION NOTE: [Compiler use guide] C Coding Guide (Using DSP)

This document explains usage and gives precautions for DSP (SH2-DSP, SH3-DSP, SH4AL-DSP), for the SuperH RISC engine C/C++ Compiler V.9.

Table of contents

1. SH-DSP Features 2

2. DSP Library 6
2.1 Summary 6
2.1.1 Data Format 7
2.1.2 Efficiency 8
2.2 Details of DSP library function 8
2.2.1 Fast Fourier transform 8
2.2.2 Window Functions 34
2.2.3 Filters 38
2.2.4 Convolution and Correlation 62
2.2.5 Other 72
2.3 Performance of the DSP Library 99

3. DSP-C Specifications 105
3.1 Fixed-Point Data Type 105
3.2 Memory Qualifier 108
3.3 Saturation Qualifier 111
3.4 Circular Qualifier 113
3.5 Type Conversion 114

1. SH-DSP Features

The SH-DSP core is provided with a DSP unit which performs 16-bit fixed-point operations and is ideal for:

- Multiply-and-accumulate operations
- Repeated processing

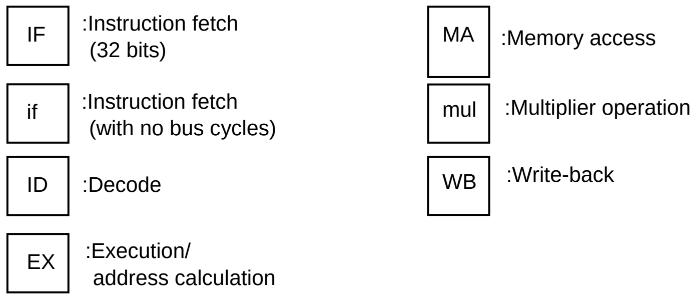
It is thus capable of performing at high speed the JPEG processing, audio processing, and filter processing required for multimedia operations.

In previous SH cores (the SH-1 core example in figure 1.1), the performance of multiply-and-accumulate operations were determined by the three cycles constituting the multiplier operation time in pipeline operation. Even if the multiplier operation time were improved to a single cycle, however, stalling of the pipeline would occur due to instruction data transfer, so that the long-term average time would be 2.5 cycles.

In the SH-DSP core, the DSP unit operation time is a single cycle, and an X bus/Y bus is provided as the data bus, so that multiply-and-accumulate operations take just one cycle (figure 1.2). Here the long-term average time is also one cycle.

Code example

```
clrmac
mac.w @r4+,@r5+
mac.w @r4+,@r5+
mac.w @r4+,@r5+
mac.w @r4+,@r5+
rts
sts   macl,r0
```



Example of pipeline operation

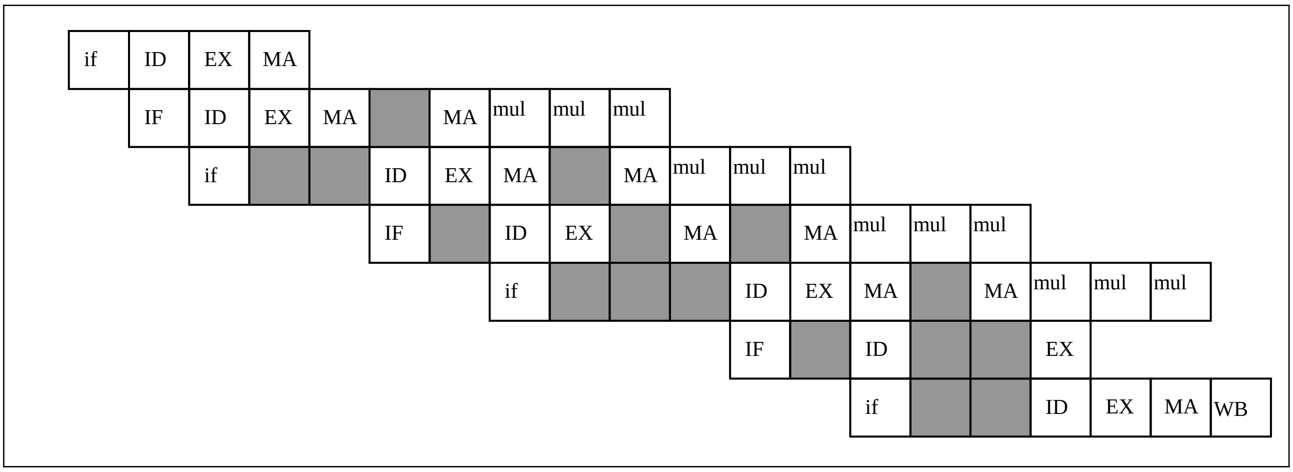
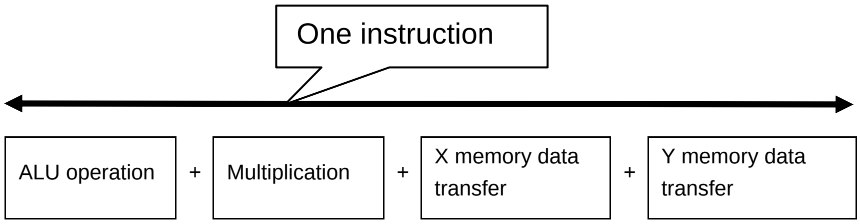


Figure 1.1 Multiple-and-Accumulate Instruction Executed in SH Core



Code example

```
Instruction 1                                         MOVX.W@R4+,X0        MOVY.W@R6+,Y0
Instruction 2                    PMULSX0,Y0,M0        MOVX.W@R4+,X1        MOVY.W@R6+,Y1
Instruction 3 PADD A0,M0,A0   PMULSX1,Y1,M1        MOVX.W@R4+,X0        MOVYW@R6+,Y0
Instruction 4 PADD A0,M1,A0   PMULS X0,Y0,M0       MOVX.W@R4+,X1        MOVYW@R6+,Y1
```

Example of pipeline operation

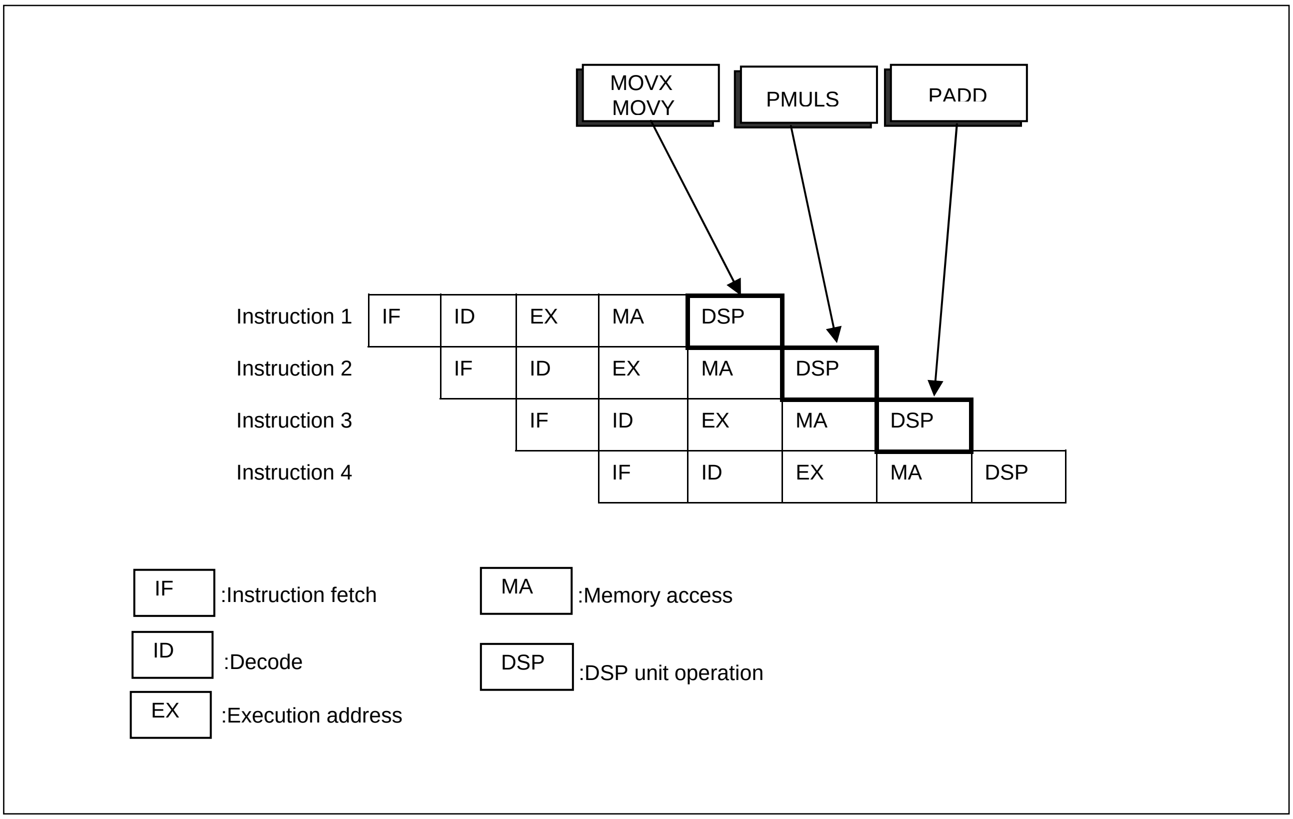


Figure 1.2 Multiply-and-Accumulate Instruction Executed in SH-DSP Core

Further, the SH-DSP core is equipped with hardware mechanisms to reduce disruption of the pipeline due to repeated processing.

In previous SH cores, conditional branching was used for loop processing. Conditional branching acts to disrupt pipelines, adding to processing overhead.

In the SH-DSP core there is a zero-overhead mechanism which reduces to zero the pipeline disruption due to this loop processing. Simply by setting the loop start and finish addresses and number of loops, loop processing is completed without performing conditional branching. Many critical software operations depend on loop processing; this is a hardware mechanism which is effective in speeding software execution.

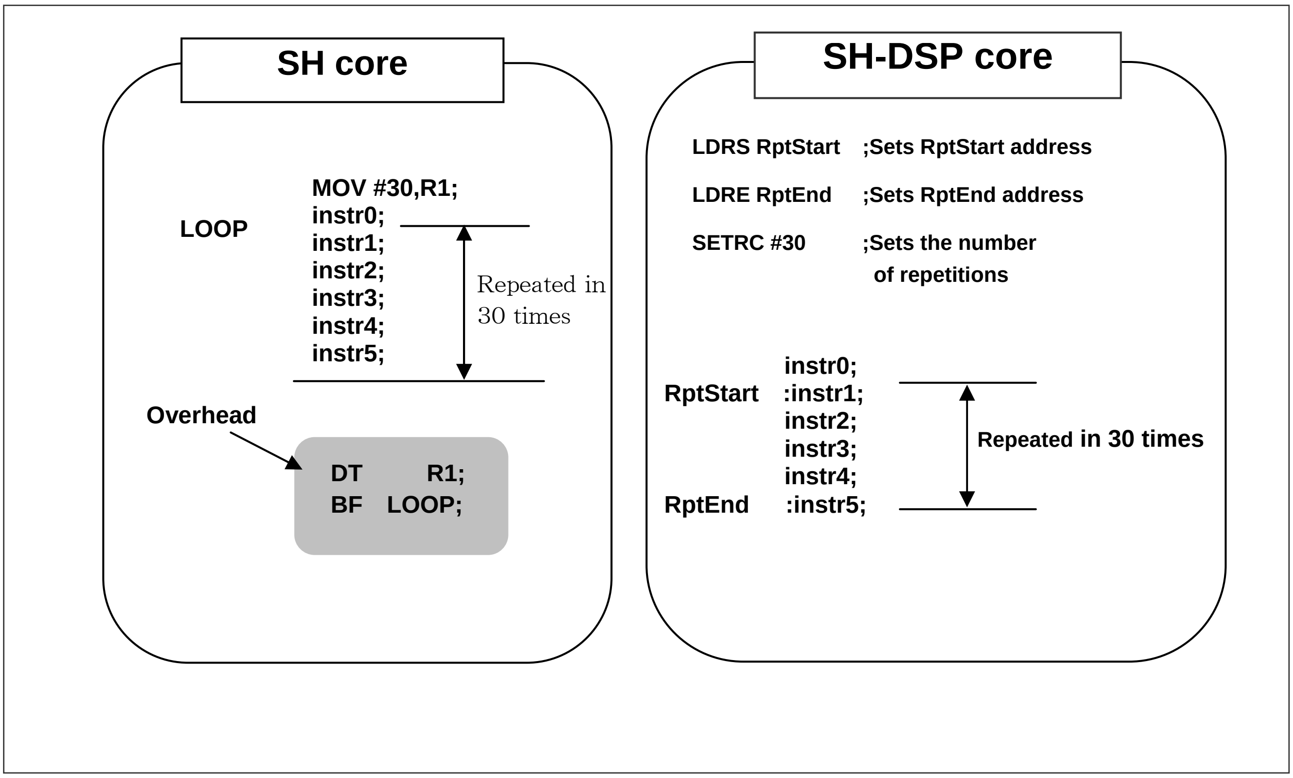


Figure 1.3 Repetition Processing

The SH-DSP core is able to execute in parallel five instructions, as shown in figure 1.4: condition evaluation, ALU operations, signed multiplication, X memory access, and Y memory access. By combining these instructions, various multiply-and-accumulate operations can be performed at high speed.

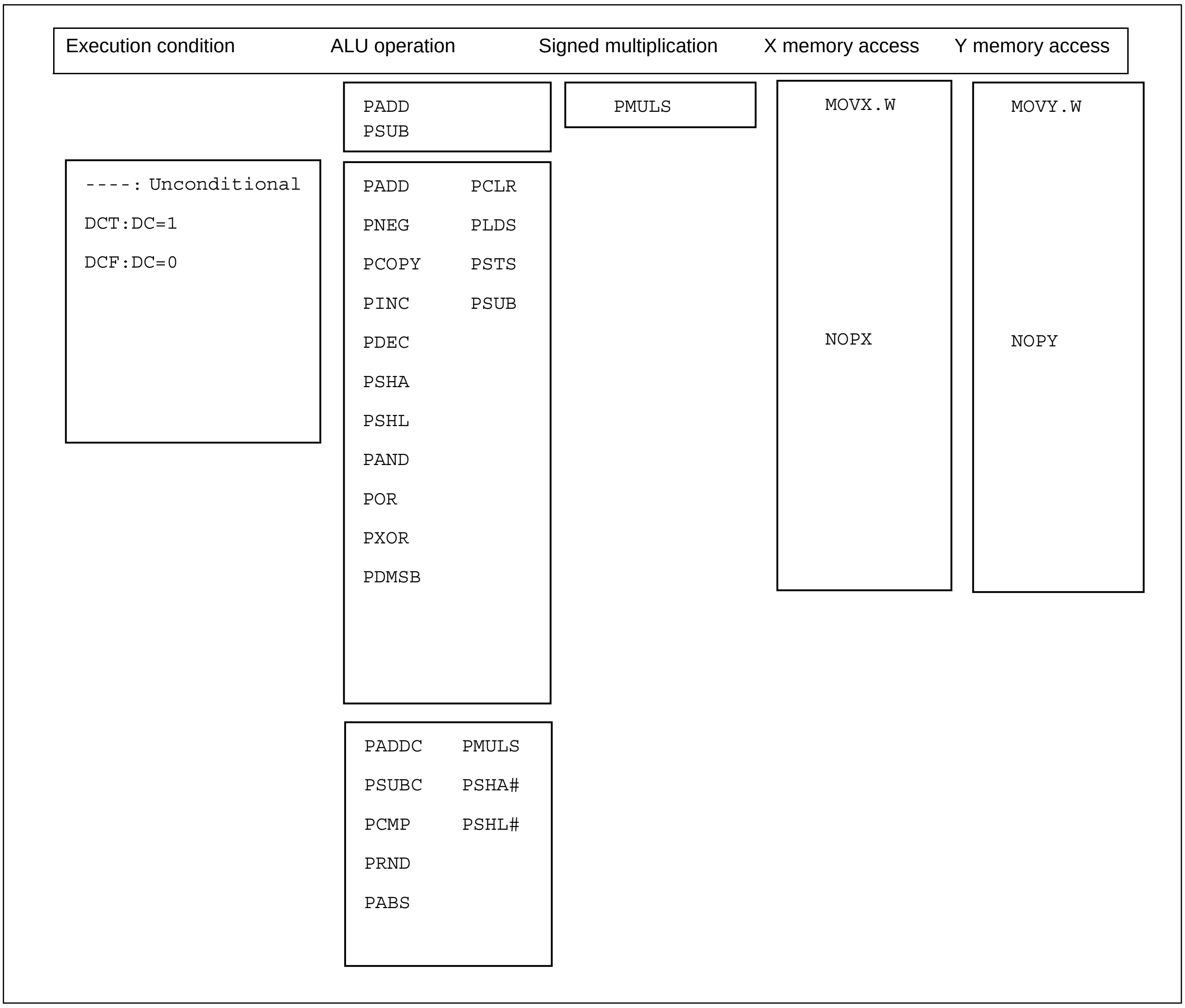


Figure 1.4 DSP Instructions (Parallel Instructions)

2. DSP Library

2.1 Summary

This section explains the digital signal processing (DSP) library that can be used with SH2-DSP and SH3-DSP (henceforward jointly referred to simply as SH-DSP) This library includes standard DSP functions, and by using them singly or consecutively, DSP operations can be performed.

This library includes the following functions.

- Fast Fourier transforms
- Window functions
- Filters
- Convolution and correlation
- Other

The functions in this library are, with the exception of fast Fourier transforms and filters, reentrant.

When using this library, include the files shown in table 2.1. In addition, as shown in table 2.2, link to the library corresponding to the CPU and compile options.

When this library is called on, if the function finishes normally, EDSP_OK is returned as the value, and if an error occurs, EDSP_BAD_ARG or EDSP_NO_HEAP is returned as the value. For the details of return values, refer to the explanation of each function.

Table 2.1 Include Files for Use with the DSP Library

Type of library	Description	Include file
DSP Library	The library performs DSP operations	<ensigdsp.h>
		<filt_ws.h>*[1]

Note: 1. When using filter functions, include them only once in the user program.

Table 2.2 DSP Library List

CPU	Option		Library Name
SH2-DSP	-pic=0		shdsplib.lib
	-pic=1		shdsppic.lib
SH3-DSP SH4AL-DSP	-pic=0	-endian=big	sh3dspnb.lib
	-pic=1	-endian=big	sh3dsppb.lib
	-pic=0	-endian=little	sh3dspnl.lib
	-pic=1	-endian=little	sh3dsppl.lib

2.1.1 Data Format

This library handles data as signed 16-bit fixed point numbers. Signed 16-bit fixed point numbers, as shown in (a) in figure 2.1, are of the data format where the point is fixed to the right side of the most significant bit (MSB), and values from -1 to $1-2^{-15}$ can be expressed.

In this library, transfer of data uses the short type of data format. Therefore, when using this library from C/C++ programs, it is necessary to express data in signed 16-bit fixed point numbers.

Example: +0.5 expressed as a signed 16-bit fixed point number is H'4000. Therefore, the short type actual parameter passed to the library function is H'4000.

Internal operations within this library use signed 32-bit fixed point numbers and signed 40-bit fixed point numbers. Signed 32-bit fixed point numbers, are of the data format as shown in (b) in figure 2.1, and values from -1 to $1-2^{-31}$ can be expressed. Signed 40-bit fixed point numbers, are of the data format with an additional 8-bit guard bit as shown in (c) in figure 2.1, and values from -2^{8} to $2^{8}-2^{-31}$ can be expressed.

The multiplication results of signed 16-bit fixed point numbers are saved as signed 32-bit fixed point numbers. With fixed point multiplication using DSP instructions, only in the case of H'8000 x H'8000 is it necessary to be careful in case overflow occurs. In addition, the least significant bit (LSB) of multiplication results is normally 0. When the multiplication results are used in the next operation, the upper 16 bits are removed, and the result is converted to a signed 16-bit fixed point number. In this case, there is a possibility that underflow or reduced accuracy may occur.

In multiply-and-accumulate operations of this library, addition results are saved as signed 40-bit fixed point numbers. Be careful that overflow does not occur when performing addition.

If an overflow occurs when performing an operation, a correct result will not be obtained. In order to prevent overflows, it is necessary to perform scaling of coefficients or of input data. Scaling functions are built into this library. For the details of scaling, refer to the explanation of each function.

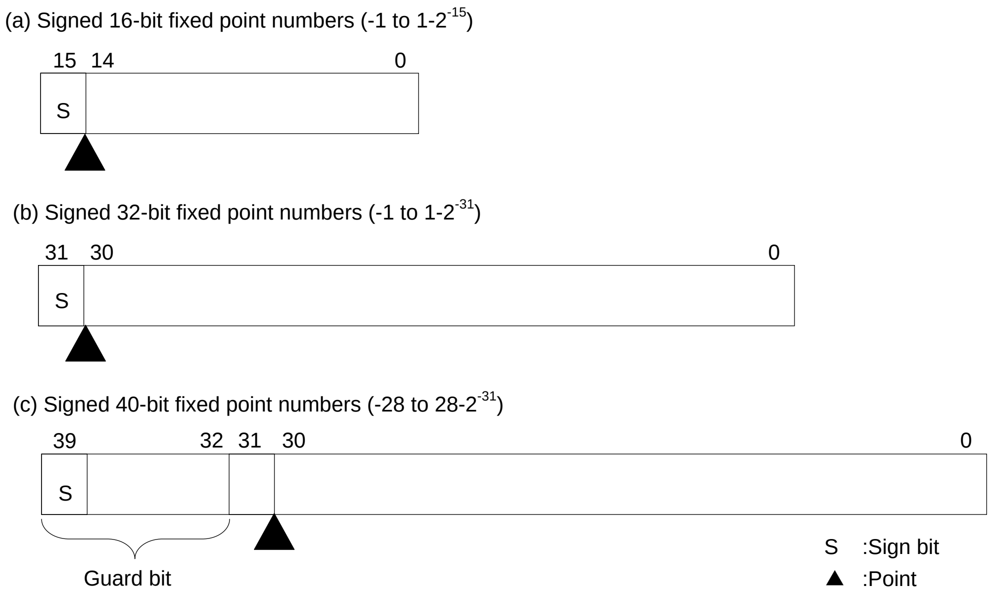


Figure 2.1 Data Format

2.1.2 Efficiency

The functions in this library are optimized to execute at high speed on SH-DSP.
In order to use the library efficiently, when deciding the memory map of the system in development, observe the following two recommendations as far as possible.

- Allocate memory that supports 32-bit read for 1 cycle for program code segments.
- Allocate memory that supports 16-bit (or 32-bit) read and write for 1 cycle for data segments.

If the microcomputer to be used has 32-bit memory built in of sufficient capacity to allocate the library code and data, it is best to allocate it to the 32-bit memory. If it is necessary to use other memory, follow the above recommendation as far as possible.

2.2 Details of DSP library function

2.2.1 Fast Fourier transform

(1) List of functions

Table 2.3 List of DSP Library Functions (Fast Fourier Transform)

No.	Type	Function Name	Description
1	not-in-place complex number FFT	FftComplex	Performs not-in-place complex number FFT
2	not-in-place real-number FFT	FftReal	Performs not-in-place real-number FFT
3	not-in-place inverse complex number FFT	IfftComplex	Performs not-in-place inverse complex number FFT
4	not-in-place inverse real-number FFT	IfftReal	Performs not-in-place inverse real-number FFT
5	in-place complex number FFT	FftInComplex	Performs in-place complex number FFT
6	in-place real number FFT	FftInReal	Performs in-place real-number FFT
7	in-place inverse complex number FFT	IfftInComplex	Performs in-place inverse complex number FFT
8	in-place inverse real-number FFT	IfftInReal	Performs in-place inverse real-number FFT
9	logarithmic absolute value	LogMagnitude	Converts complex number data into logarithmic absolute values
10	FFT rotation factor generation	InitFft	Generates FFT rotation factors
11	FFT rotation factor release	FreeFft	Releases the memory used to store FFT rotation factors

Note: For details on not-in-place and in-place, refer to "(5) FFT structure".

The factors use the scaling defined by the user to execute forward direction high speed Fourier transforms and reverse direction high speed Fourier transforms.
Forward direction Fourier transforms are defined using the following equations.

$$y_n = 2^{-s} \sum_{n=0}^{N} e^{-2j\pi n/N} \cdot x_n$$

Here, s represents the number of stages for performing scaling, and N represents the number of data elements.
Reverse direction Fourier transforms are defined using the following equations.

$$y_n = 2^{-s} \sum_{n=0}^{N} e^{2j\pi n/N} \cdot x_n$$

For details on scaling, refer to “(4) Scaling”.

(2) Complex number data array format

FFT and IFFT complex number data arrays are allocated to X memory for real numbers and to Y memory for imaginary numbers. However, the allocation of real number FFT output data and real number IFFT input data differs. If the arrays in which real numbers and imaginary numbers are stored are defined as x and y respectively, the real number component of the DC component goes into x[0], and rather than the imaginary number component of the DC component, the real number component of the Fs/2 component goes into y[0] (the DC component and Fs/2 component are both real numbers, and the imaginary number component is 0).

(3) Real number data array format

There are 3 kinds of FFT and IFFT real number data array formats as follows.

- Stored in a single array, and allocated to an arbitrary memory block.
- Stored in a single array, and allocated to X memory.
- Divided into 2 arrays for storage. The size of each array is N/2, and the first half of the array is allocated to X memory, and the second half is allocated to Y memory.

Only the first specification method is available for FftReal. The user can select the second or third methods for IfftReal, FftInReal, and IfftInReal.

(4) Scaling

The signal strength of base 2 FFT doubles at each stage, and peak signal amplitude also doubles. For this reason, when converting to a high intensity signal there is a possibility that overflows may occur. However, by halving the signal at each stage (this is called ‘scaling’), overflows can be prevented. However, if excessive scaling is implemented, there is a possibility that unnecessary quantization noise may occur.

The optimal balance of scaling between overflows and quantization noise depends greatly on the characteristics of the input signals. In order to prevent overflows with spectra with large peaks in the signals, maximum scaling is necessary, but with impulse signals, scaling is hardly required at all.

Performing scaling at every stage is the safest method. If the intensity of the input data is less than 2^{30}, overflows can be prevented using this method. With this library, scaling can be specified for each stage. Therefore, by specifying scaling precisely, the impact of overflows and quantization noise can be suppressed to the minimum.

In order to specify the method of scaling, each FFT function parameter includes ‘scale’. ‘scale’ corresponds to each stage from the least significant bit to each individual bit. If the corresponding scale bit is set to 1, at every stage, division by 2 is executed.

In order to increase execution speed, base 4 FFT is used in this library. ‘scale’ corresponds to each stage from the least significant bit to each two bits. If either one bit is set to 1, division by 2 is executed. If both bits are set to 1, division by 4 is executed. In other words, this is the same as if two base 2 FFT stages are replaced with one base 4 FFT stage. However, with base 4 FFT, there is a greater possibility that quantization noise will occur than with base 2 FFT.

An example of ‘scale’ is shown below.

- When scale = H’FFFFFFFF (or size-1), scaling is performed for all base 2 FFT stages. If the intensity of all the input data is less than 2^{30}, overflow will not occur.
- When scale = H’55555555, scaling is performed for every other base 2 FFT stage.
- When scale = 0, scaling is not performed.

These scale values are defined as ensigdsp.h, EFFTALLSCALE(H'FFFFFFFF), EFFTMIDSCALE(H'55555555), and EFFTNOSCALE(0)

(5) FFT structure

The FFT structures of this library are of 2 kinds, not-in-place FFT, and in-place FFT

With not-in-place FFT, the input data is removed from RAM, FFT is executed, and the output result is stored in another place in RAM specified by the user.

On the other hand, with in-place FFT, the input data is removed from RAM, FFT is executed, and the output result is stored in the same place in RAM. If this method is used, execution time for the FFT is increased, but the memory space used can be decreased.

When using other FFT functions with input data, use not-in-place FFT. In addition, when seeking to conserve memory space, use in-place FFT.

(6) Explanation of each function

(a) not-in-place complex number FFT

- **Description:**
- Format:

```
int FftComplex (short op_x[], short op_y[],
                const short  ip_x[], const short ip_y[], long size, long scale)
```

- Parameters:

`op_x[]`	Real number component of output data
`op_y[]`	Imaginary number component of output data
`ip_x[]`	Real number component of input data
`ip_y[]`	Imaginary number component of input data
`size`	FFT size
`scale`	Scaling specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 4` •`size is not a power of 2` •`size > max_fft_size`

- Explanation of this function:

Executes a complex number fast Fourier transform.

- Remarks:

As this function performs not-in-place, provide input arrays and output arrays separately. For details on allocation of complex number data arrays, refer to “(2) Complex number data array format”. Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size. For details on scaling, refer to “(4) Scaling”. ‘scale’ uses the lower $\log_2$ (size) bit. This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <math.h>      Include header
#include <ensigdsp.h>
```

```
#define MAX_FFT_SAMP   64
#define MIN_CFFT_SIZE    4
long ip_scale=0xffffffff;
long size = MIN_CFFT_SIZE;

#pragma section X
short ip_x[MAX_FFT_SAMP];
short op_x[MAX_FFT_SAMP];
#pragma section Y
short ip_y[MAX_FFT_SAMP];
short op_y[MAX_FFT_SAMP];
#pragma section

/* Data for cycle counting */
#define TWOPI 6.283185307 /* data */
void main()
{
    int i,j;
    long n_samp;

    n_samp=MAX_FFT_SAMP; /* data */
    for (j = 0; j < n_samp; j++){
        ip_x[j] = cos(j * TWOPI/n_samp) * 8188;
        ip_y[j] = sin(j * TWOPI/n_samp) * 8188;
    }
    if(InitFft(n_samp) != EDSP_OK){
        printf("Initfft != err end");
    }
    if(FftComplex(op_x,op_y,ip_x,ip_y,n_samp,EFFTALLSCALE) != EDSP_OK){
        printf("FftComplex error¥n");
    }
    FreeFft();
    for(i=0;i<n_samp;i++){
        printf("[%d] op_x=%d  op_y=%d  ¥n",i,op_x[i],op_y[i]);
    }
}
```

Variables placed in X or Y memory are defined by a pragma section within the section.

Variables placed in X or Y memory are defined by a pragma section within the section.

Data creation for FFT

FFT initialization function;
Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2.

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

(b) not-in-place real number FFT

Description:

- Format:

```
int FftReal (short op_x[], short op_y[], const short ip[],
             long size, long scale)
```

- Parameters:

`op_x[]`	Real number component of positive output data
`op_y[]`	Imaginary number component of positive output data
`ip[]`	Real number input data
`size`	FFT size
`scale`	Scaling specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 8` •`size is not a power of 2` •`size > max_fft_size`

- Explanation of this function:

Executes a real number fast Fourier transform.

- Remarks:

size/2 positive output data is stored in op_x and op_y. Negative output data is the conjugate complex number of positive output data. In addition, as the values of output data of 0 and $F_S/2$ are real numbers, the real number output with $F_S/2$ is stored in op_y[0].
As this function performs not-in-place, provide input arrays and output arrays separately.
For details on allocation of complex number and real number data arrays, refer to "(2) Complex number data array format" and "(3) Real number data array format".
Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size.
For details on scaling, refer to "(4) Scaling".
'scale' uses the lower $\log_2$ (size) bit.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define    VLEN  64
#define    TWOPI 6.28318530717959
/* global data declarations */
#pragma section X
short output_x[VLEN];
#pragma section Y
short output_y[VLEN];
#pragma section
void main()
{
    short i;
    int k;
    short input[VLEN];
    short output[VLEN];
/* generate two sinusoids */
    k = VLEN / 8;
    for (i = 0; i < VLEN; i++)
       input[i] = floor(16383 * cos(TWOPI * k * i / VLEN) + 0.5);
    k = VLEN * 3 / 8;
    for (i = 0; i < VLEN; i++)
        input[i] += floor(16383 * cos(TWOPI * k * i / VLEN) + 0.5);

/* do FFT */
    if (InitFft(VLEN) != EDSP_OK)
        printf("InitFft problem¥n");
    if (FftReal(output_x, output_y, input, VLEN, EFFTALLSCALE) != EDSP_OK)
        printf("FftReal problem¥n");
    FreeFft();
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Creation of data for FFT

FFT initialization function;

Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2.

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

(c) not-in-place inverse complex number FFT

Description:

- Format:

```
int IfftComplex (short op_x[], short op_y[],
                 short ip_x[], const short ip_y[],
                 long size, long scale)
```

- Parameters:

`op_x[]`	Real number component of output data
`op_y[]`	Imaginary number component of output data
`ip_x[]`	Real number component of input data
`ip_y[]`	Imaginary number component of input data
`size`	Inverse FFT size
`scale`	Scaling specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 4` •`size is not a power of 2` •`size > max_fft_size`

- Explanation of this function:

Executes a complex number inverse fast Fourier transform.

- Remarks:

As this function performs not-in-place, provide input arrays and output arrays separately.
For details on allocation of complex number data arrays, refer to “(2) Complex number data array format”.
Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size.
For details on scaling, refer to “(4) Scaling”.
‘scale’ uses the lower log2 (size) bit.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
```

Include header

```
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */
long ip_scale=8188;
```

Variables placed in X or Y memory are defined by a pragma section within the section.

```
#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* input array */
short opi_x[MAX_IFFT_SIZE];          /* normal output array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
short opi_y[MAX_IFFT_SIZE];
#pragma section
void main()
{
    int i,j;
    long scale;
    long max_size;
    max_size=MAX_IFFT_SIZE;/* data */
```

Creation of data for FFT (data used to execute FftComplex)

```
    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
        ipi_y[j] = sin(j * TWOPI/max_size) * ip_scale;
    }
```

FFT initialization function;

Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2.

```
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  ¥n");
    }
    else {
        if(FftInComplex(ipi_x, ipi_y, max_size,EFFTALLSCALE) != EDSP_OK){
            printf("FftInComplex err  end  ¥n");
        }
        for (j = 0; j < max_size; j++){
            opi_x[j]=0;
            opi_y[j]=0;
        }
```

This processing performs FFT calculations and uses the results as input values for an inverse FFT function; normally it is not necessary.

```
        if(IfftComplex(opi_x, opi_y, ipi_x, ipi_y, max_size,
                                    EFFTALLSCALE)!= EDSP_OK){
             printf("IfftComplex err  end  ¥n");
         }
         for (j = 0; j < max_size; j++){
            printf("[%d]  opi_x=%d  op_y=%d ¥n",j, opi_x[j],opi_y[j]);
         }
         FreeFft();
     }
}
```

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

(d) not-in-place real number inverse FFT

Description:

- Format:

```
int IfftReal (short op_x[], short scratch_y[],
              const short ip_x[], const short ip_y[], long size,
              long scale, int op_all_x)
```

- Parameters:

op_x[]	Real number output data
scratch_y[]	Scratch memory or real number output data
ip_x[]	Real number component of positive input data
ip_y[]	Imaginary number component of positive input data
size	Inverse FFT size
scale	Scaling specification
op_all_x	Allocation specification of output data

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	In any of the following cases •size < 8 •size is not a power of 2 •size > max_fft_size •op_all_x ≠ 0 or 1

- Explanation of this function:

Executes a real number inverse fast Fourier transform.

- Remarks:

Store size/2 positive input data in ip_x and ip_y. Negative input data is the conjugate complex number of positive input data. In addition, as the values of input data of 0 and $F_S/2$ are real numbers, store the real number input with $F_S/2$ in ip_y[0].|
The format of output data is specified with op_all_x. If op_all_x=1, all output data is stored in op_x. If op_all_x=0, the first size/2 output data is stored in op_x, and the remainder of the size/2 output data is stored in scratch_y.
As this function performs not-in-place, provide input arrays and output arrays separately.
For details on allocation of complex number and real number data arrays, refer to “(2) Complex number data array format” and “(3) Real number data array format”.
Store size/2 data in ip_x and ip_y respectively. size or size/2 data is stored in op_x depending on the value of op_all_x.
Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size.
For details on scaling, refer to “(4) Scaling”.
‘scale’ uses the lower $\log_2$ (size) bit.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
```

Include header

```
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data *//
long ip_scale=8188;
#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* input array */
short opi_x[MAX_IFFT_SIZE];          /* normal output array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
short opi_y[MAX_IFFT_SIZE];
#pragma section
```

Variables placed in X or Y memory are defined by a pragma section within the section.

```
void main()
{
    int i,j;
    long scale;
    long max_size;
    max_size=MAX_IFFT_SIZE;/* data */
    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
    }
```

Creation of data for FFT (data used to execute FftReal)

```
    if (InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  ¥n");
    }
```

FFT initialization function;

Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2. Also required for inverse FFT.

```
    else {
        if(FftInReal(ipi_x, ipi_y, max_size,EFFTALLSCALE,1) != EDSP_OK){
            printf("FftInReal err  end  ¥n");
        }
if(IfftReal(opi_x, opi_y, ipi_x, ipi_y, max_size, EFFTALLSCALE,1)!=
   EDSP_OK){
            printf("IfftReal err end  ¥n");
        }
        for (j = 0; j < max_size; j++){
           printf("[%d]  opi_x=%d  op_y=%d ¥n",j, opi_x[j],opi_y[j]);
        }
```

This processing performs FFT calculations and uses the results as input values for an inverse FFT function; normally it is not necessary.

```
        FreeFft();
    }
}
```

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

(e) in-place complex number FFT

Description:

- Format:

```
int FftInComplex (short data_x[], short data_y[],
                  long size, long scale)
```

- Parameters:

`data_x[]`	Real number component of input data
`data_y[]`	Imaginary number component of input and output data
`size`	FFT size
`scale`	Scaling specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 4` •`size is not a power of 2` •`size > max_fft_size`

- Explanation of this function:

Executes an in-place complex number fast Fourier transform.

- Remarks:

For details on allocation of complex number data arrays, refer to “(2) Complex number data array format”.
Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size.
For details on scaling, refer to “(4) Scaling”.
‘scale’ uses the lower $\log_2$ (size) bit.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
```

Include header

```
#define MAX_FFT_SAMP  64
#define TWOPI 6.283185307 /* data */
long ip_scale=0xffffffff;

#pragma section X
short ip_x[MAX_FFT_SAMP];
#pragma section Y
short ip_y[MAX_FFT_SAMP];
#pragma section
```

Variables placed in X or Y memory are defined by a pragma section within the section.

```
void main()
{
    int i,j;
    long max_size;
    long n_samp;
    n_samp=MAX_FFT_SAMP;
    max_size=n_samp;/* data */
    for (j = 0; j < n_samp; j++){
        ip_x[j] = cos(j * TWOPI/n_samp) * ip_scale;
        ip_y[j] = sin(j * TWOPI/n_samp) * ip_scale;
    }
```

Data creation for FFT

```
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error¥n");
    }
```

FFT initialization function;

Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2.

```
    if(FftInComplex(ip_x, ip_y, n_samp,EFFTALLSCALE ) != EDSP_OK){
        printf("FftInComplex error¥n");
    }
    FreeFft();
```

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

```
    for(i=0;i<max_size;i++){
        printf("[%d] ip_x=%d  ip_y=%d  ¥n",i,ip_x[i],ip_y[i]);
    }
}
```

(f) in-place real number FFT

Description:

- Format:

```
int FftInReal (short data_x[], short data_y[], long size,
               long scale, int ip_all_x)
```

- Parameters:

`data_x[]`	Real number data when input, and real number component of the positive output data when output
`data_y[]`	Real number data or unused for input, and imaginary number component of the positive output data when output
`size`	FFT size
`scale`	Scaling specification
`ip_all_x`	Allocation specification of input data

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 8` •`size is not a power of 2` •`size > max_fft_size` •`ip_all_x ≠ 0 or 1`

- Explanation of this function:

Executes an in-place real number fast Fourier transform.

- Remarks:

The format of input data is specified with ip_all_x. If ip_all_x=1, all input data is removed from data_x. If ip_all_x=0, the first half of size/2 input data is removed from data_x, and the second half of size/2 input data is removed from data_y.
After execution of this function, size/2 positive output data is stored in data_x and data_y. Negative output data is the conjugate complex number of positive output data. In addition, as the values of output data of 0 and $F_S/2$ are real numbers, the real number output with $F_S/2$ is stored in data_y[0].
For details on allocation of complex number and real number data arrays, refer to "(2) Complex number data array format" and "(3) Real number data array format".
Store size/2 data in data_y. size or size/2 data is stored in data_x depending on the value of ip_all_x.
Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size.
For details on scaling, refer to "(4) Scaling".
'scale' uses the lower $\log_2$ (size) bit.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_FFT_SAMP  64
#define TWOPI 6.283185307 /* data */
long ip_scale=8188;
/*long ip_scale=0xffffffff;*/

#pragma section X
short ip_x[MAX_FFT_SAMP];
#pragma section Y
short ip_y[MAX_FFT_SAMP];
#pragma section


void main()
{
    int i,j;
    long max_size;
    long n_samp;
    int ip_all_x;
    n_samp=MAX_FFT_SAMP;
    max_size=n_samp;/* data */

    for (j = 0; j < n_samp; j++){
        ip_x[j] = cos(j * TWOPI/n_samp) * ip_scale;
        ip_y[j] = 0;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error¥n");
    }

    ip_all_x = 1;
    if(FftInReal(ip_x, ip_y, n_samp,EFFTALLSCALE ,ip_all_x) != EDSP_OK){
        printf("FftInReal error¥n");
    }
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Data creation for FFT

FFT initialization function;

Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2.

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

```
    FreeFft();
    for(i=0;i<max_size;i++){
        printf("[%d] ip_x=%d  ip_y=%d  ¥n",i,ip_x[i],ip_y[i]);
    }
}
```

(g) in-place complex number inverse FFT

Description:

- Format:

```
int IfftInComplex (short data_x[], short data_y[],
                   long size, long scale)
```

- Parameters:

`data_x[]`	Real number component of input data
`data_y[]`	Imaginary number component of input and output data
`size`	Inverse FFT size
`scale`	Scaling specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 4` •`size is not a power of 2` •`size > max_fft_size`

- Explanation of this function:

Executes an in-place complex number inverse fast Fourier transform.

- Remarks:

For details on allocation of complex number data arrays, refer to “(2) Complex number data array format”.
Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size.
For details on scaling, refer to “(4) Scaling”.
‘scale’ uses the lower $\log_2$ (size) bit.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
```

Include header

```
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */
```

Variables placed in X or Y memory are defined by a pragma section within the section.

```
long ip_scale=8188;
#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* input array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
#pragma section

void main()
{
    int i,j;
    long scale;
    long max_size;
```

Data creation for FFT (data used as input for FftInComplex)

```
    max_size=MAX_IFFT_SIZE;/* data */
    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
        ipi_y[j] = sin(j * TWOPI/max_size) * ip_scale;
    }
```

FFT initialization function;
Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2.
Also required for inverse FFT.

```
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  ¥n");
    }
    else {
        if(FftInComplex(ipi_x, ipi_y, max_size,EFFTALLSCALE) != EDSP_OK){
            printf("FftInComplex err  end  ¥n");
        }
        if(IfftInComplex(ipi_x, ipi_y, max_size,EFFTALLSCALE) != EDSP_OK){
            printf("IfftInComplex err  end  ¥n");
        }
```

This processing performs FFT calculations and uses the results as input values for an inverse FFT function; normally it is not necessary.

```
        for (j = 0; j < max_size; j++){
           printf("[%d]  ipi_x=%d  ip_y=%d ¥n",j, ipi_x[j],ipi_y[j]);
```

```
            }
            FreeFft();
        }
    }
```

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

(h) in-place real number inverse FFT

Description:

- Format:

```
int IfftInReal (short data_x[], short data_y[], long size,
                long scale, int op_all_x)
```

- Parameters:

`data_x[]`	Real number component of positive input data when input, and real number data when output
`data_y[]`	Imaginary number component of positive input data when input, and real number data when output or unused
`size`	Inverse FFT size
`scale`	Scaling specification
`op_all_x`	Allocation specification of output data

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 8` •`size is not a power of 2` •`size > max_fft_size` •`op_all_x ≠ 0 or 1`

- Explanation of this function:

Executes an in-place real number inverse fast Fourier transform.

- Remarks:

Store size/2 positive input data in data_x and data_y. Negative input data is the conjugate complex number of positive input data. In addition, as the values of input data of 0 and $F_S/2$ are real numbers, store the real number input with $F_S/2$ in data_y[0].
The format of output data is specified with op_all_x. If op_all_x=1, all output data is stored in data_x. If op_all_x=0, the first half of the size/2 output data is stored in data_x, and the second half of the size/2 output data is stored in data_y.
For details on allocation of complex number and real number data arrays, refer to “(2) Complex number data array format” and “(3) Real number data array format”.
Store size/2 data in data_y. size or size/2 data is stored in data_x depending on the value of op_all_x.
Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size.
For details on scaling, refer to “(4) Scaling”.
‘scale’ uses the lower $\log_2$ (size) bit.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */

long ip_scale=8188;

#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* input array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
#pragma section
void main()
{
    int i,j;
    long scale;
    long max_size;
    max_size=MAX_IFFT_SIZE;/* data */
    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
    }
    if(InitFft(max_size) != EDSP_OK){
           printf("InitFft error end  ¥n");
    }
    else {
        if(FftInReal(ipi_x, ipi_y, max_size,EFFTALLSCALE,1) != EDSP_OK){
            printf("FftInReal err  end  ¥n");
        }
        if(IfftInReal(ipi_x, ipi_y, max_size, EFFTALLSCALE,1) != EDSP_OK){
            printf("IfftInReal err  end  ¥n");
        }
        for (j = 0; j < max_size; j++){
           printf("[%d]  ipi_x=%d  ip_y=%d ¥n",j, ipi_x[j],ipi_y[j]);
        }
        FreeFft();
    }
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Data creation for FFT (data used as input for FftInReal)

FFT initialization function;

Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2.

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

(i) Logarithmic absolute value

Description:

- Format:

```
int LogMagnitude (short output[], const short ip_x[],
                  const short ip_y[], long no_elements,
                  float fscale)
```

- Parameters:

`output[]`	Real number output z
`ip_x[]`	Input real number component x
`ip_y[]`	Input imaginary number component y
`no_elements`	Number of output data elements N
`fscale`	Output scaling coefficient

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases
	•`no_elements < 1`
	•`no_elements > 32767`
	•$\lvert fscale \rvert > 2^{15}/(10\log_{10}2^{31})$

- Explanation of this function:

Calculates the logarithmic absolute value of complex number input data in decibel units, and writes the scaling results in the output array.

- Remarks:

$z(n)=10fscale \cdot \log_{10}(x(n)^2+y(n)^2) \quad 0 \le n < N$

For details on allocation of complex number data arrays, refer to “(2) Complex number data array format”.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */
long ip_scale=8188;
#pragma section X
short ipi_x[MAX_IFFT_SIZE];            /* input array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
#pragma section
void main()
{
    int i,j;
    long scale;
    long max_size;
    short output[MAX_IFFT_SIZE];
    max_size=MAX_IFFT_SIZE;/* data */

    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  ¥n");
    }
    else {
        if(FftInReal(ipi_x, ipi_y, max_size,EFFTALLSCALE,1) != EDSP_OK){
            printf("FftInReal err  end  ¥n");
        }
        if(LogMagnitude(output, ipi_x,ipi_y, max_size/2, 2) != EDSP_OK){
            printf("LogMagnitude err  end  ¥n");
        }
        for (j = 0; j < max_size/2; j++){
           printf("[%d]  output=%d  ¥n",j, output[j]);
        }
        FreeFft();
    }
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Data creation for FFT

FFT function;
Creates data used by the LogMagnitude function.

This frees the table used in FFT calculations.

If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft. This is not directly related to LogMagnitude.

(j) Rotation factor generation

Description:

- Format:
  `int InitFft (long max_size)`

- Parameters:

`max_size`	Maximum size of the required FFT

- Returned value:

`EDSP_OK`	Successful
`EDSP_NO_HEAP`	The memory space that can be obtained with malloc is insufficient
`EDSP_BAD_ARG`	In any of the following cases •`max_size < 2` •`max_size is not a power of 2` •`max_size > 32,768`

- Explanation of this function:

Generates the rotation factor (1/4 size) to be used by the FFT function.

- Remarks:

The rotation factor is stored in the memory obtained by malloc.
When the rotation factor is generated, the max_fft_size global variable is updated. max_fft_size shows the maximum capacity size of the FFT.
Be sure to call on this function once before calling on the first FFT function.
Make max_size 8 or more.
The rotation factor is generated by the conversion size specified by max_size. Use the same rotation factor when executing a FFT function with a smaller size than max_size.
The address of the rotation factor is stored inside the internal variable. Do not access this with the user program.
This function is not reentrant.

(k) Rotation factor release

Description:

- Format:
  `void FreeFft (void)`

- Parameters:
  None

- Returned value:
  None

- Explanation of this function:

Releases the memory used to store the rotation factors.

- Remarks:

Make the max_fft_size global variable 0. When executing the FFT function again after executing FreeFft, be sure to execute InitFft first.
This function is not reentrant.

2.2.2 Window Functions

(1) List of functions

Table 2.4 DSP Library Function List (Window Functions)

No.	Type	Function Name	Description
1	Blackman window	GenBlackman	Generates a Blackman window.
2	Hamming window	GenHamming	Generates a Hamming window.
3	Hanning window	GenHanning	Generates a Hanning window.
4	Triangular window	GenTriangle	Generates a triangular window.

(2) Explanation of each function

(a) Blackman window

Description:

- Format:
  `int GenBlackman (short output[], long win_size)`

- Parameters:

`output[]`	Output data W(n)
`win_size`	Window size N

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	win_size ≤ 1

- Explanation of this function:

Generates a Blackman window, and outputs to output.

- Remarks:

Use VectorMult when applying this window to actual data.
The function to be used is shown below.

$$W(n) = \left(2^{15} - 1\right)\left[0.42 - 0.5\cos\left(\frac{2\pi n}{N}\right) + 0.08\cos\left(\frac{4\pi n}{N}\right)\right] \quad 0 \le n < N$$

Example of use:

```
#include <stdio.h>       } Include header
#include <ensigdsp.h>
#define MAXN     10
void main()
{
    int i;
    long len;
    short output[MAXN];
    len=MAXN ;
    if(GenBlackman(output, len) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<len;i++){
        printf("output=%d ¥n",output[i]);
    }
}
```

(b) Hamming window

Description:

- Format:
  `int GenHamming (short output[], long win_size)`

- Parameters:

`output[]`	Output data W(n)
`win_size`	Window size N

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	win_size ≤ 1

- Explanation of this function:

Generates a Hamming window, and outputs to output.

- Remarks:

Use VectorMult when applying this window to actual data.
The function to be used is shown below.

$$W(n) = \left(2^{15} - 1\right)\left[0.54 - 0.46\cos\left(\frac{2\pi n}{N}\right)\right] \quad 0 \le n < N$$

Example of use:

```
#include <stdio.h>      } Include header
#include <ensigdsp.h>
#define MAXN  10
void main()
{
    int i;
    long len;
    short output[MAXN];
    len=MAXN ;
    if(GenHamming(output, len) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<len;i++){
        printf("output=%d ¥n",output[i]);
    }
}
```

(c) Hanning window

Description:

- Format:
  `int GenHanning (short output[], long win_size)`

- Parameters:

`output[]`	Output data W(n)
`win_size`	Window size N

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	win_size ≤ 1

- Explanation of this function:

Generates a Hanning window, and outputs to output.

- Remarks:

Use VectorMult when applying this window to actual data.

The function to be used is shown below.

$$W(n) = \left(\frac{2^{15} - 1}{2}\right)\left[1 - \cos\left(\frac{2\pi n}{N}\right)\right] \quad 0 \le n < N$$

Example of use:

```
#include <stdio.h>      } Include header
#include <ensigdsp.h>
#define MAXN  10
void main()
{
    int i;
    long len;
    short output[MAXN];
    len=MAXN ;
    if(GenHanning(output, len) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<len;i++){
        printf("output=%d ¥n",output[i]);
    }
}
```

(d) Triangular window

Description:

- Format:
`int GenTriangle (short output[], long win_size)`

- Parameters:

output[]	Output data W(n)
win_size	Window size N

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	win_size ≤ 1

- Explanation of this function:

Generates a triangular window, and outputs to output.

- Remarks:

Use VectorMult when applying this window to actual data.
The function to be used is shown below.

$$W(n) = \left(2^{15} - 1\right)\left[1 - \left|\frac{2n - N + 1}{N + 1}\right|\right] \quad 0 \le n < N$$

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>        Include header
#define MAXN     10
void main()
{
    int i;
    long len;
    short output[MAXN];
    len=MAXN ;
    if(GenTriangle(output, len) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<len;i++){
        printf("output=%d ¥n",output[i]);
    }
}
```

2.2.3 Filters

(1) List of functions

Table 2.5 DSP Library Function List (Filters)

No.	Type	Function Name	Description
1	FIR	Fir	Performs finite impulse-response filter processing
2	FIR for single data elements	Fir1	Performs finite impulse-response filter processing for a single data element
3	IIR	Iir	Performs infinite impulse-response filter processing
4	IIR for single data elements	Iir1	Performs infinite impulse-response filter processing for a single data element
5	Double precision IIR	Diir	Performs double-precision infinite impulse-response filter processing
6	Double precision IIR for single data elements	Diir1	Performs double-precision infinite impulse-response filter processing for a single data element
7	Adaptive FIR	Lms	Performs adaptive FIR filter processing
8	Adaptive FIR for single data elements	Lms1	Performs adaptive FIR filter processing for a single data element
9	FIR work space allocation	InitFir	Allocates a work space for use by the FIR filter
10	IIR work space allocation	InitIir	Allocates a work space for use by the IIR filter
11	Double precision IIR work space allocation	InitDIir	Allocates a work space for use by the DIIR filter
12	Adaptive FIR work space allocation	InitLms	Allocates a work space for use by the LMS filter
13	FIR work space release	FreeFir	Releases the work space allocated by InitFir
14	IIR work space release	FreeIir	Releases the work space allocated by InitIir
15	Double precision IIR work space release	FreeDIir	Releases the work space allocated by InitDIir
16	Adaptive FIR work space release	FreeLms	Releases the work space allocated by InitLms

Note: When using any of these functions, include filt_ws.h only once in the user program.

(2) Coefficient scaling

When executing filter processing, there is a possibility that saturation or quantization noise may occur. These can be suppressed to the minimum by performing scaling of these filter coefficients. However, it is necessary to perform scaling giving careful consideration to the impact of saturation and quantization noise. If the coefficient is too large there is a possibility that saturation may occur. If it is too small, quantization noise may occur.

With the FIR (finite impulse response) filter, saturation will not occur if the filter coefficient is set so that the following equation is applied.

coeff[i] ≠ H’8000 (for all instances of i)

Σ|coeff| < 224

res_shift = 24

coeff is the filter coefficient, and res_shift is the number of bits shifted to the right at output.

However, when there are many input signals, even if a smaller res_shift value is used (or a bigger coeff value), the possibility of saturation is slight, and quantization noise can be reduced by a wide margin. In addition, if there is a possibility that the input value includes H’8000, set all coeff values to be in the range of H’8001 to H’7FFF.

The IIR (infinite impulse response) filter has a recursive structure. For this reason, the scaling method explained above is not suitable.

The LMS (least mean square) adaptive filter is the same as the FIR filter. However, when adapting the coefficient, there may be cases where saturation occurs. In this case, make the settings so that H’8000 is not included in the coefficient.

(3) Work space

With digital filters, there is information that must be saved between one process and the next. This information is stored in memory that can be accessed with the minimum of overhead. With this library, the Y-RAM area is used as the work space. Before executing filter processing, call on the Init function and initialize the work space.

The work space memory is accessed by the library function. Do not access the work space directly from the user program.

(4) Using memory

In order to use SH-DSP efficiently, allocate filter coefficients to X memory. Input and output data can be allocated to arbitrary memory segments.

Allocate filter coefficients to X memory using the #pragma section instruction.

Each filter is allocated to the work space from the global buffer using the Init function. The global buffer is allocated to Y memory.

(5) Explanation of each function

(a) FIR

Description:

- Format:

```
int Fir (short output[], const short input[], long no_samples,
         const short coeff[], long no_coeffs, int res_shift,
         short *workspace)
```

- Parameters:

`output[]`	Output data y
`input[]`	Input data x
`no_samples`	Number of input data elements N
`coeff[]`	Filter coefficient h
`no_coeffs`	Number of coefficients (filter length) K
`res_shift`	Right shift applied to each output.
`workspace`	Pointer to the work space

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_samples < 1` •`no_coeffs ≤ 2` •`res_shift < 0` •`res_shift > 25`

- Explanation of this function:

Performs finite impulse-response (FIR) filter processing

- Remarks:

The latest input data is saved in the work space. The results of filter processing of input are written to output.

$$y(n) = \left[\sum_{k=0}^{K-1} h(k)\ x(n-k)\right] \cdot 2^{-res\_shift}$$

The results of multiply-and-accumulate operations are saved as 39 bits. Output y(n) is the lower 16 bits fetched from the res_shift bit right shifted results. When an overflow occurs, this is the positive or negative maximum value.
For details on coefficient scaling, refer to “(2) Coefficient scaling”.
Before calling on this function, call on InitFir, and initialize the work space of the filter.
If the same array is specified for output as for input, input will be overwritten.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
#define NFN 8 /* number of functions */
#define FIL_COUNT   32  /* number of data objects */
#define N   32

#pragma section X
static short coeff_x[FIL_COUNT];
#pragma section
short data[FIL_COUNT] =  {
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,};
short coeff[8] =  {
         0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000,};

void main()
{
    short *work, i;
    short output[N];
    int nsamp, ncoeff, rshift;
    /* copy coeffs into X RAM */
    for(i=0;i<NFN;i++) {
        coeff_x[i] = coeff[i];/* Sets coefficient */
    }
    for (i = 0; i < N; output[i++] = 0) ;
    ncoeff = NFN;/* Sets the number of coefficients  */
    nsamp = FIL_COUNT;/* set number of samples */
    rshift = 12;
    if (InitFir(&work, ncoeff) != EDSP_OK){
        printf("Init Problem¥n");
    }
    if(Fir(output, data, nsamp, coeff_x, ncoeff, rshift, work) != EDSP_OK){
        printf("Fir Problem¥n");
    }
    if (FreeFir(&work, ncoeff) != EDSP_OK){
        printf("Free Problem¥n");
    }
    for(i=0;i<nsamp;i++){
        printf("#%2d output:%6d ¥n",i,output[i]);
    }
}
```

Include header

Set the filter coefficients in X memory. Since Y memory is used by the library as the work area to calculate filter coefficients, Y memory should not be used.

Set filter coefficients in X memory as variables.

Filter initialization:
(1) Work area address
(2) Number of coefficients
This is necessary before Fir function execution. The work area in Y memory uses (number of coefficients)*2+8 bytes.

The FreeFir function frees the work area used for Fir calculations; The FreeFir function must always be performed after Fir execution. If this function is not executed, memory resources are wasted.

(b) FIR for single data elements

Description:

- Format:

```
int Fir1 (short *output, short input, const short coeff[],
          long no_coeffs, int res_shift, short *workspace)
```

- Parameters:

`output`	Pointer to output data y(n)
`input`	Input data x(n)
`coeff[]`	Filter coefficient h
`no_coeffs`	Number of coefficients (filter length) K
`res_shift`	Right shift applied to each output.
`workspace`	Pointer to the work space

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_coeffs ≤ 2` •`res_shift < 0` •`res_shift > 25`

- Explanation of this function:

Performs finite impulse-response (FIR) filter processing for single data elements.

- Remarks:

The latest input data is saved in the work space. The results of filter processing of input are written to *output.

$$y(n) = \left[\sum_{k=0}^{K-1} h(k)\ x(n-k)\right] \cdot 2^{-res\_shift}$$

The results of multiply-and-accumulate operations are saved as 39 bits. Output y(n) is the lower 16 bits fetched from the res_shift bit right shifted results. When an overflow occurs, this is the positive or negative maximum value.
For details on coefficient scaling, refer to “(2) Coefficient scaling”.
Before calling on this function, call on InitFir, and initialize the work space of the filter.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
#define NFN 8  /* number of functions */
#define MAXSH 25
#define N   32
#pragma section X
static short coeff_x[NFN];
#pragma section
short data[32] =  {
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400};
short coeff[8] =  {
          0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000};
void main()
{
    short *work, i;
    short output[N];
    int ncoeff, rshift;

    /* copy coeffs into X RAM */
    for(i=0;i<NFN;i++) {
        coeff_x[i] = coeff[i];/* Sets coefficient */
    }
    for (i = 0; i < N; output[i++] = 0) ;
    rshift = 12;
    ncoeff = NFN;/* Sets the number of coefficients */
    if (InitFir(&work, NFN) != EDSP_OK){
        printf("Init Problem¥n");
    }
    for(i=0;i<N;i++) {
        if(Fir1(&output[i], data[i], coeff_x, ncoeff, rshift, work) !=
    EDSP_OK){
            printf("Fir1 Problem¥n");
        }
        printf(" output[%d]=%d ¥n",i,output[i]);
    }
    if (FreeFir(&work, NFN) != EDSP_OK){
        printf("Free Problem¥n");
    }
}
```

Include header

Set the filter coefficients in X memory. Since Y memory is used by the library as the work area to calculate filter coefficients, Y memory should not be used.

Set filter coefficients in X memory as variables.

Filter initialization:
(1) Work area address
(2) Number of coefficients
This is necessary before Fir1 function execution. The work area in Y memory uses (number of coefficients)*2+8 bytes.

Fir1 means that the number of data elements that are set to the Fir function is 1. When executing Fir1 multiple times, the Init Fir and FreeFir functions must be executed before and after the for statement respectively.

(c) IIR

Description:

- Format:

```
int Iir (short output[], const short input[], long no_samples,
         const short coeff[], long no_sections, short *workspace)
```

- Parameters:

`output[]`	Output data y_{K-1}
`input[]`	Input data x_0
`no_samples`	Number of input data elements N
`coeff[]`	Filter coefficient
`no_sections`	Number of secondary filter sections K
`workspace`	Pointer to the work space

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_samples < 1` •`no_sections < 1` •a_{0k} < 0 •a_{0k} > 16

- Explanation of this function:

Performs infinite impulse-response (IIR) filter processing.

- Remarks:

This filter is configured whereby a secondary filter, or biquad, is linked to the K number tandem. Additional scaling is performed with the output of each biquad. The filter coefficient is specified with a signed 16-bit fixed point number
The output of each biquad is subject to the following equation.

$d_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2_{15}x(n)] \cdot 2^{-15}$

$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)] \cdot 2^{-a0k}$

The input x_k (n) for k is the output y_{k-1} (n) of the previous section. The input of the first section (k=0) is read from input. The output of the last section (k=K-1) is written to output.
Set coeff in the following order of coefficients.

a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}

The a_{0k} item is the number of bits for right shift to be performed on the output of the biquad for k.
Each biquad performs a 32-bit multiply-and-accumulate operation. The output of each biquad is the lower 16 bits fetched from the 15-bit or a_{0k} right shifted results. When an overflow occurs, this is the positive or negative maximum value.
Before calling on this function, call on InitIir, and initialize the work space of the filter.
If the same array is specified for output as for input, input will be overwritten.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>

#define K  4
#define NUMCOEF  (6*K)
#define N 50
#pragma section X
static short coeff_x[NUMCOEF];
#pragma section
static short coeff[24] = {15, 19144, -7581,  5301, 10602,  5301,
                          15, -1724,-23247, 13627, 27254, 13627,
                          15, 19144, -7581,  5301, 10602,  5301,
                          15, -1724,-23247, 13627, 27254, 13627};

static short input[50] = {32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000 };
void main()
{
    short *work, i;
    short output[N];

    for(i=0;i<NUMCOEF;i++) {
        coeff_x[i] = coeff[i];
    }
    if (InitIir(&work, K) != EDSP_OK){
        printf("Init Problem¥n");
    }
    if (Iir(output, input, N, coeff_x, K, work) !=
        printf("EDSP_OK not returned¥n");
    }
    if (FreeIir(&work, K) != EDSP_OK){
        printf("Free Problem¥n");
    }
    for(i=0;i<N;i++){
        printf("#%2d output:%6d ¥n",i,output[i]);
    }
}
```

Include header

Set the filter coefficients in X memory. Since Y memory is used by the library as the work area to calculate filter coefficients, Y memory should not be used.

Six filter coefficients should be set in one section. The leading element in a section is the number of right-shifts, and is not a filter coefficient.

Set filter coefficients in X memory as variables.

Filter initialization:
(1) Work area address
(2) Number of filter sections
This is necessary before Iir function execution. The work area in Y memory uses ((number of filter sections)*2*2) bytes.

The FreeIir function frees the work area used for Iir calculations; The FreeIir function must always be performed after Iir execution. If this function is not executed, memory resources are wasted.

(d) IIR for single data elements

Description:

- Format:

```
int Iir1 (short *output, short input, const short coeff[],
          long no_sections, short *workspace)
```

- Parameters:

`output`	Pointer to output data $y_{K-1}(n)$
`input`	Input data x_0 (n)
`coeff[]`	Filter coefficient
`no_sections`	Number of secondary filter sections K
`workspace`	Pointer to the work space

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_sections < 1` •a_{ok} < 0 •a_{ok} > 16

- Explanation of this function:

Performs infinite impulse-response (IIR) filter processing for single data elements.

- Remarks:

This filter is configured whereby a secondary filter, or biquad, is linked to the K number tandem. Additional scaling is performed with the output of each biquad. The filter coefficient is specified with a signed 16-bit fixed point number.
The output of each biquad is subject to the following equation.

$d_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2^{15}x(n)] \cdot 2^{-15}$

$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)] \cdot 2^{-a0k}$

The input x_k (n) for k is the output y_{k-1} (n) of the previous section. The input of the first section (k=0) is read from input.
The output of the last section (k=K-1) is written to output.
Set coeff in the following order of coefficients.
a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}
The a0k item is the number of bits for right shift to be performed on the output of the biquad for k.
Each biquad performs a 32-bit saturation operation. The output of each biquad is the lower 16 bits fetched from the 15-bit or a_{0k} right shifted results. When an overflow occurs, this is the positive or negative maximum value.
Before calling on this function, call on InitIir, and initialize the work space of the filter.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
#define K    4
#define NUMCOEF  (6*K)
#define N    50
#pragma section X
static short coeff_x[NUMCOEF];
#pragma section
static short coeff[24] = {15, 19144, -7581,  5301, 10602,  5301,
                              15, -1724,-23247, 13627, 27254, 13627,
                              15, 19144, -7581,  5301, 10602,  5301,
                              15, -1724,-23247, 13627, 27254, 13627};

static short input[50] = {32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000 };
short keisu[5]={ 1,2,20,4,5 };
```

Include header

Set the filter coefficients in X memory. Since Y memory is used by the library as the work area to calculate filter coefficients, Y memory should not be used.

Six filter coefficients should be set in one section. The leading element in a section is the number of right-shifts, and is not a filter coefficient.

```
void main()
{
    short *work, i;
    short output[N];
    for(i=0;i<NUMCOEF;i++) {
        coeff_x[i] = coeff[i];
     }
    if (InitIir(&work, K) != EDSP_OK){
        printf("Init Problem¥n");
    }
    for(i=0;i<N;i++){
        if (Iir1(&output[i], input[i], coeff_x, K, work) != EDSP_OK){
            printf("EDSP_OK not returned¥n");
        }
        printf("output[%d]:%d ¥n" ,i,output[i]);
    }
    if (FreeIir(&work, K) != EDSP_OK){
        printf("Free Problem¥n");
    }
}
```

Set filter coefficients in X memory as variables.

Filter initialization:
(1) Work area address
(2) Number of filter sections
This is necessary before Iir1 function execution. The work area in Y memory uses (number of filter sections)*2*2 bytes.

Iir1 means that the number of data elements that are set to the Iir function is 1. When executing Iir1 multiple times, the Init Iir and FreeIir functions must be executed before and after the for statement.

(e) Double precision IIR

Description:

- Format:

```
int DIir (short output[], const short input[], long no_samples,
          const long coeff[], long no_sections, long *workspace)
```

- Parameters:

`output[]`	Output data y_{K-1}
`input[]`	Input data x
`no_samples`	Number of input data elements N
`coeff[]`	Filter coefficient
`no_sections`	Number of secondary filter sections K
`workspace`	Pointer to the work space

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_samples < 1` •`no_sections < 1` •$a_{0k} < 3$ •$k < K-1$ and $a_{0k} > 32$ •$k = K-1$ and $a_{0k} > 48$

- Explanation of this function:

Performs double-precision infinite impulse-response filter processing

- Remarks:

This filter is configured whereby a secondary filter, or biquad, is linked to the K number tandem. Additional scaling is performed with the output of each biquad. The filter coefficient is specified with a signed 32-bit fixed point number.
The output of each biquad is subject to the following equation.
$d_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2^{29}x(n)] \cdot 2^{-31}$
$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)] \cdot 2^{-a0k} \cdot 2^{2}$
The input x_k (n) for k is the output y_{k-1} (n) of the previous section. The input of the first section (k=0) is read from the 16-bit left shifted value of input. The output of the last section (k=K-1) is written to output.
Set coeff in the following order of coefficients.
a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}
The a_{0k} item is number of bits for right shift to be performed on the output of the biquad for k.
DIir differs from Iir in that the filter coefficient is specified with a 32-bit value rather than a 16-bit value. The results of multiply-and-accumulate operations are saved as 64 bits. The output of intermediate stages is the lower 32 bits fetched from the a_{0k} bit right shifted results. When an overflow occurs, this is the positive or negative maximum value. At the last stage, the lower 16 bits are fetched from the a_{0k-1} bit right shifted results. When an overflow occurs, this is the positive or negative maximum value.
Before calling on this function, call on InitDIir, and initialize the work space of the filter.
The delayed node d_k (n) is rounded off to 30 bits, and when an overflow occurs, this is the positive or negative maximum value.
When using DIir, specify the coefficient with a signed 32-bit fixed point number. In this case, when a_{0k} is k < K-1 set it as 31, and when k=K-1 set it as 47.
As the speed of execution of Iir is faster than that of DIir, if double precision calculation is required, use Iir.
If the same array is specified for output as for input, input will be overwritten.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <filt_ws.h>
#include <ensigdsp.h>
#define K  5
#define NUMCOEF  (6*K)
#define N  50
#pragma section X
static long coeff_x[NUMCOEF];
#pragma section
static long coeff[60] =
      {31,1254686956, -496866304, 347415747, 694831502, 347415746,
       31,-113001278,-1523568505, 893094203,1786188388, 893094206,
       31,1254686956, -496866304, 347415747, 694831502, 347415746,
       31,-113001278,-1523568505, 893094203,1786188388, 893094206,
       47,1254686956, -496866304, 347415747, 694831502, 347415746};

static short input[100] = {
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000 };

void main()
{
    short i;
    short output[N];
    long *work;
    long nsamp;

    for(i=0;i<NUMCOEF;i++)
        coeff_x[i] = coeff[i];
    if(InitDIir(&work,K) != EDSP_OK){
        printf("InitDIir Problem¥n");
    }
    if(DIir(output, input, N, coeff_x, K, work) != EDSP_OK){
        printf("DIir Problem¥n");
    }
    if(FreeDIir(&work, K) != EDSP_OK){
        printf("FreeDIir Problem¥n");
    }
    for(i=0;i<N;i++){
        printf("output[%d]=%d¥n",i,output[i]);
    }
}
```

Include header

Set the filter coefficients in X memory.

Since Y memory is used by the library as the work area to calculate filter coefficients, Y memory should not be used.

Six filter coefficients should be set in one section. The leading element in a section is the number of right-shifts, and is not a filter coefficient.

The number of right-shifts is 31 except for the last section; the last section is 47.

Set filter coefficients in X memory as variables.

Filter initialization:
(1) Work area address
(2) Number of filter sections
This is necessary before DIir function execution. The work area in Y memory uses (number of filter sections)*4*2 bytes.

The FreeDIir function frees the work area used for DIir calculations; The FreeDIir function must always be performed after DIir execution. If this function is not executed, memory resources are wasted.

(f) Double precision IIR for single data elements

Description:

- Format:

```
int DIir1  (short output[], const short input[], long no_samples,
            const long coeff[], long no_sections,
            long *workspace)
```

- Parameters:

`output`	Pointer to output data $y_{K-1}(n)$
`input`	Input data x_0 (n)
`coeff[]`	Filter coefficient
`no_sections`	Number of secondary filter sections K
`workspace`	Pointer to the work space

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_sections < 1` •$a_{0k} < 3$ •$k < K-1$ and $a_{0k} > 32$ •$k = K-1$ and $a_{0k} > 48$

- Explanation of this function:

Performs double precision infinite impulse-response filter processing for single data elements.

- Remarks:

This filter is configured whereby a secondary filter, or biquad, is linked to the K number tandem. Additional scaling is performed with the output of each biquad. The filter coefficient is specified with a signed 32-bit fixed point number.
The output of each biquad is subject to the following equation.

$d_k(n)=[a_{1k}d_k(n-1)+a_{2k}dk(n-2)+2^{29}x(n)] \cdot 2^{-31}$

$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)] \cdot 2^{-a0k} \cdot 2^2$

The input x_k (n) for k is the output y_{k-1} (n) of the previous section. The input of the first section (k=0) is read from the 16-bit left shifted value of input. The output of the last section (k=K-1) is written to output.
Set coeff in the following order of coefficients.

a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}

The a_{0k} item is number of bits for right shift to be performed on the output of the biquad for k.
DIir1 differs from Iir1 in that the filter coefficient is specified with a 32-bit value rather than a 16-bit value. The results of multiply-and-accumulate operations are saved as 64 bits. The output of intermediate stages is the lower 32 bits fetched from the a_{0k} bit right shifted results. When an overflow occurs, this is the positive or negative maximum value. At the last stage, the lower 16 bits are fetched from the a_{0k-1} bit right shifted results. When an overflow occurs, this is the positive or negative maximum value.
Before calling on this function, call on InitDIir, and initialize the work space of the filter.
The delayed node d_k (n) is rounded off to 30 bits, and when an overflow occurs, this is the positive or negative maximum value.
When using DIir1, specify the coefficient with a signed 32-bit fixed point number. In this case, when a_{0k} is k < K-1 set it as 31, and when k=K-1 set it as 47.
As the speed of execution of Iir1 is faster than that of DIir1, if double precision calculation is required, use Iir1.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
#define K   5
#define NUMCOEF  (6*K)
#define N   50
#pragma section X
static long coeff_x[NUMCOEF];
#pragma section
static long coeff[60] =
 {31,1254686956, -496866304, 347415747, 694831502, 347415746,
  31,-113001278,-1523568505, 893094203,1786188388, 893094206,
  31,1254686956, -496866304, 347415747, 694831502, 347415746,
  31,-113001278,-1523568505, 893094203,1786188388, 893094206,
  47,1254686956, -496866304, 347415747, 694831502, 347415746};
static short input[N]    = {32000, 32000, 32000, 32000, 32000,
                               32000, 32000, 32000, 32000, 32000,
                               32000, 32000, 32000, 32000, 32000,
                               32000, 32000, 32000, 32000, 32000,
                               32000, 32000, 32000, 32000, 32000,
                               32000, 32000, 32000, 32000, 32000,
                               32000, 32000, 32000, 32000, 32000,
                               32000, 32000, 32000, 32000, 32000,
                               32000, 32000, 32000, 32000, 32000,
                               32000, 32000, 32000, 32000, 32000 };
void main()
{
    short i;
    short output[N];
    long *work;

    for(i=0;i<NUMCOEF;i++)
        coeff_x[i] = coeff[i];
    if(InitDIir(&work, K) != EDSP_OK){
        printf("Init Problem¥n");
    }
    for(i=0;i<N;i++){
        if(DIir1(&output[i], input[i], coeff_x, K, work) !=EDSP_OK){
            printf("DIir1 error¥n");
        }
        printf("output[%d]:%d ¥n" ,i,output[i]);
    }
    if(FreeDIir(&work, K) != EDSP_OK){
        printf("Free DIir error¥n");
    }
}
```

Include header

Six filter coefficients should be set in one section. The leading element in a section is the number of right-shifts, and is not a filter coefficient.

Set the filter coefficients in X memory. Since Y memory is used by the library as the work area to calculate filter coefficients, Y memory should not be used.

The number of right-shifts is 31 except for the last section; the last section is 47.

Set filter coefficients in X memory as variables.

Filter initialization:
(1) Work area address
(2) Number of filter sections
This is necessary before DIir1 function execution. The work area in Y memory uses (number of filter sections)*4*2

DIir1 means that the number of data elements that are set to the DIir function is 1. When executing DIir1 multiple times, the InitDIir and FreeDIir functions must be executed before and after the for statement respectively.

(g) Adaptive FIR

Description:

- Format:

```
int Lms  (short output[], const short input[],
          const short ref_output[], long no_samples,
          short coeff[], long no_coeffs, int res_shift,
          short conv_fact, short  *workspace)
```

- Parameters:

`output[]`	Output data y
`input[]`	Input data x
`ref_output[]`	Desired output value d
`no_samples`	Number of input data elements N
`coeff[]`	Adaptive filter coefficient h
`no_coeffs`	Number of coefficients K
`res_shift`	Right shift applied to each output
`conv_fact`	Convergence coefficient 2μ
`workspace`	Pointer to the work space

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_samples < 1` •`no_coeffs ≤ 2` •`res_shift < 0` •`res_shift > 25`

- Explanation of this function:

Using a least mean square (LMS) algorithm, executes real number adaptive FIR filter processing.

- Remarks:

FIR filters are defined using the following equations.

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k)\ x(n-k)\right] \cdot 2^{-res\_shift}$$

The results of multiply-and-accumulate operations are saved as 39 bits. Output y(n) is the lower 16 bits fetched from the res_shift bit right shifted results. When an overflow occurs, this is the positive or negative maximum value.
Update of filter coefficients is performed using the Widrow-Hoff algorithm.
$h_{n+1}(k)=h_n(k)+2\mu e(n)x(n-k)$
Here, e(n) is the margin of error between the desired signal and the actual output.
$e(n)=d(n)-y(n)$
With the $2\mu e(n)x(n-k)$ calculation, multiplication of 16 bits x 16 bits is performed 2 times. The upper 16 bits of both multiplication results are saved, and when an overflow occurs, this is the positive or negative maximum value. If the value of the updated coefficient is H'8000, there is a possibility that overflow may occur with the multiply-and-accumulate operation. Set the value of the coefficient to be in the range of H'8001 to H'7FFF.
For details on coefficient scaling, refer to "(2) Coefficient scaling". As the coefficient is adapted using an LMS filter, the safest scaling method is to set less than 256 coefficients and to set res_shift to 24.
conv_fact should normally be set to positive. Do not set it to H'8000.
Before calling on this function, call on InitLms, and initialize the filter.
If the same array is specified for output as for input or for ref_output, input or ref_output will be overwritten.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
```

Include header

Set the filter coefficients in X memory. Since Y memory is used by the library as the work area to calculate filter coefficients, Y memory should not be used.

```
#define K    8
#define N    40
#define TWOMU 32767
#define RSHIFT 15
#define MAXSH 25
#pragma section X
static short coeff_x[K];
#pragma section
short data[N] =  {
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400};
short coeff[K] =  {
         0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000};

static short ref[N] = { -107, -143, 998, 1112, -5956,
                             -10781, 239, 13655, 11202, 2180,
                             -687, -2883, -7315, -6527, 196,
                             4278, 3712, 3367, 4101, 2703,
                             591, 695, -1061, -5626, -4200,
                             3585, 9285, 11796, 13416, 12994,
                             10231, 5803, -449, -6782, -11131,
                             -10376, -2968, 2588, -1241, -6133};
void main()
{
    short *work, i, errc;
    short output[N];
    short twomu;
    int nsamp, ncoeff, rshift;

    /* copy coeffs into X RAM */
    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    nsamp = 10;
    ncoeff = K;
    rshift = RSHIFT;
    twomu = TWOMU;
    for (i = 0; i < N; output[i++] = 0) ;
    ncoeff = K;/* Sets the number of coefficients */
    nsamp = N;/* Sets the number of samples */
```

Set filter coefficients in X memory as variables.

```
for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    if (InitLms(&work, K) != EDSP_OK){
        printf("Init Problem¥n");
    }
    if(Lms(output, data, ref, nsamp, coeff_x, ncoeff, RSHIFT,TWOMU, work) !=
   EDSP_OK){
        printf("Lms Problem¥n");
    }
    if (FreeLms(&work, K) != EDSP_OK){
        printf( "Free Problem¥n");
    }
    for (i = 0; i < N; i++){
        printf("#%2d output:%6d ¥n",i,output[i]);
    }
}
```

Filter initialization:
(1) Work area address
(2) Number of coefficients
This is necessary before LMS function execution. The work area in Y memory uses (number of coefficients)*2+8 bytes.

The FreeLms function frees the work area used for Lms calculations; the FreeLms function must always be executed after Lms execution. If this function is not executed, memory resources are wasted.

(h) Adaptive FIR for single data elements

Description:

- Format:

```
int Lms1  (short *output, short input, short ref_output,
           short coeff[], long no_coeffs, int res_shift,
           short conv_fact, short  *workspace)
```

- Parameters:

`output`	Pointer to output data y(n)
`input`	Input data x (n)
`ref_output`	Desired output value d(n)
`coeff[]`	Adaptive filter coefficient h
`no_coeffs`	Number of coefficients K
`res_shift`	Right shift applied to each output.
`conv_fact`	Convergence coefficient 2μ
`workspace`	Pointer to the work space

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_coeffs` ≤ 2 •`res_shift` < 0 •`res_shift` > 25

- Explanation of this function:

Using a least mean square (LMS) algorithm, executes real number adaptive FIR filter processing for single data elements.

- Remarks:

FIR filters are defined using the following equation.

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k)x(n-k)\right] \cdot 2^{-res\_shift}$$

The results of multiply-and-accumulate operations are saved as 39 bits. Output y(n) is the lower 16 bits fetched from the res_shift bit right shifted results. When an overflow occurs, this is the positive or negative maximum value.
Update of filter coefficients is performed using the Widrow-Hoff algorithm.
$h_{n+1}(k)=h_n(k)+2\mu e(n)x(n-k)$
Here, e(n) is the margin of error between the desired signal and the actual output.
e(n)=d(n)-y(n)
With the 2μe(n)x(n-k) calculation, multiplication of 16 bits x 16 bits is performed 2 times. The upper 16 bits of both multiplication results are saved, and when an overflow occurs, this is the positive or negative maximum value. If the value of the updated coefficient is H’8000, there is a possibility that overflow may occur with the multiply-and-accumulate operation. Set the value of the coefficient to be in the range of H'8001 to H'7FFF.
For details on coefficient scaling, refer to “(2) Coefficient scaling”. As the coefficient is adapted using an LMS filter, the safest scaling method is to set less than 256 coefficients and to set res_shift to 24.
conv_fact should normally be set to positive. Do not set it to H’8000.
Before calling on this function, call on InitLms, and initialize the filter.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
#define K    8
#define N    40
#define TWOMU 32767
#define RSHIFT 15
#define MAXSH 25

#pragma section X
static short coeff_x[K];
#pragma section
short data[N] =  {
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400};
short coeff[K] =  {
         0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000};
static short ref[N] = { -107, -143, 998, 1112, -5956,
                             -10781, 239, 13655, 11202, 2180,
                             -687, -2883, -7315, -6527, 196,
                             4278, 3712, 3367, 4101, 2703,
                             591, 695, -1061, -5626, -4200,
                             3585, 9285, 11796, 13416, 12994,
                             10231, 5803, -449, -6782, -11131,
                             -10376, -2968, 2588, -1241, -6133};
void main()
{
    short *work, i, errc;
    short output[N];
    short twomu;
    int nsamp, ncoeff, rshift;
    /* copy coeffs into X RAM */

    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    nsamp = 10;
    ncoeff = K;
    rshift = RSHIFT;
    twomu = TWOMU;
    for (i = 0; i < N; output[i++] = 0) ;

    ncoeff = K;/* Sets the number of coefficients */
    nsamp = N;/* Sets the number of samples */
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Set filter coefficients in X memory as variables.

```
    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    if (InitLms(&work, K) != EDSP_OK){
        printf("Init Problem¥n");
    }
    for(i=0;i<nsamp;i++){
        if(Lms1(&output[i], data[i], ref[i], coeff_x,
                              ncoeff, RSHIFT, TWOMU, work) != EDSP_OK){
            printf("Lms1 Problem¥n");
        }
    }
    if (FreeLms(&work, K) != EDSP_OK){
        printf( "Free Problem¥n");
    }
    for (i = 0; i < N; i++){
        printf("#%2d output:%6d  ¥n",i,output[i]);
    }
}
```

Filter initialization:
(1) Work area address
(2) Number of coefficients
This is necessary before LMS1 function execution. The work area in Y memory uses (number of coefficients)*2+8 bytes.

The FreeLms function frees the work area used for Lms calculations; The FreeLms function must always be performed after Lms execution. If this function is not executed, memory resources are wasted.

(i) FIR work space allocation

Description:

- Format:

```
int InitFir  (short **workspace, long no_coeffs)
```

- Parameters:

workspace	Pointer to the work space
no_coeffs	Number of coefficients K

- Returned value:

EDSP_OK	Successful
EDSP_NO_HEAP	The memory space that can be used by the work space is insufficient
EDSP_BAD_ARG	no_coeffs $\leq$ 2

- Explanation of this function:

Allocates the work space to be used by Fir and Fir1.

- Remarks:

Initializes all previously input data to 0.
Only Fir, Fir1, Lms or Lms 1 can operate the work space allocated with InitFir. Do not access the work space directly from the user program.
This function is not reentrant.

(j) IIR work space allocation

Description:

```
int InitIir  (short **workspace, long no_sections)
```

- Parameters:

workspace	Pointer to the work space
no_sections	Number of secondary filter sections K

- Returned value:

EDSP_OK	Successful
EDSP_NO_HEAP	The memory space that can be used by the work space is insufficient
EDSP_BAD_ARG	no_sections < 1

- Explanation of this function:

Allocates the work space to be used by Iir and Iir1.

- Remarks:

Initializes all previously input data to 0.
Only Iir and Iir1 can operate the work space allocated with InitIir. Do not access the work space directly from the user program.
This function is not reentrant.

(k) Double precision IIR work space allocation

Description:

- Format:
  `int InitDIir  (long **workspace, long no_sections)`

- Parameters:

`workspace`	Pointer to the work space
`no_sections`	Number of secondary filter sections K

- Returned value:

`EDSP_OK`	Successful
`EDSP_NO_HEAP`	The memory space that can be used by the work space is insufficient
`EDSP_BAD_ARG`	no_sections < 1

- Explanation of this function:

Allocates the work space to be used by DIir and DIir1.

- Remarks:

Initializes all previously input data to 0.
Only DIir and DIir1 can operate the work space allocated with InitDIir.
This function is not reentrant.

(l) Adaptive FIR work space allocation

Description:

- Format:
  `int InitLms  (short **workspace, long no_coeffs)`

- Parameters:

`workspace`	Pointer to the work space
`no_coeffs`	Number of coefficients K

- Returned value:

`EDSP_OK`	Successful
`EDSP_NO_HEAP`	The memory space that can be used by the work space is insufficient
`EDSP_BAD_ARG`	`no_coeffs ≤ 2`

- Explanation of this function:

Allocates the work space to be used by Lms and Lms1.

- Remarks:

Initializes all previously input data to 0.
Only Fir, Fir1, Lms or Lms 1 can operate the work space allocated with InitLms. Do not access the work space directly from the user program.
This function is not reentrant.

(m) FIR work space release

Description:

- Format:
  `int FreeFir  (short **workspace, long no_coeffs)`

- Parameters:

`workspace`	Pointer to the work space
`no_coeffs`	Number of coefficients K

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	`no_coeffs ≤ 2`

- Explanation of this function:

Releases the work space allocated by InitFir

- Remarks:

This function is not reentrant.

(n) IIR work space release

Description:

- Format:
  `int FreeIir  (short **workspace, long no_sections)`

- Parameters:

`workspace`	Pointer to the work space
`no_sections`	Number of secondary filter sections K

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	`no_sections < 1`

- Explanation of this function:

Releases the work space allocated by InitIir

- Remarks:

This function is not reentrant.

(o) Double precision IIR work space release

Description:

- Format:
  `int FreeDIir  (long **workspace, long no_sections)`

- Parameters:

`workspace`	Pointer to the work space
`no_sections`	Number of secondary filter sections K

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	`no_section ≤ 2`

- Explanation of this function:

Releases the work space memory allocated by InitDIir.

- Remarks:

This function is not reentrant.

(p) Adaptive FIR work space release

Description:

- Format:
  `int FreeLms  (short **workspace, long no_coeffs)`

- Parameters:

`workspace`	Pointer to the work space
`no_coeffs`	Number of coefficients K

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	`no_coeffs < 1`

- Explanation of this function:

Releases the work space memory allocated by InitLms

- Remarks:

This function is not reentrant.

2.2.4 Convolution and Correlation

(1) List of functions

Table 2.6 List of DSP Library Functions (Convolution)

No.	Type	Function Name	Description
1	Complete convolution	ConvComplete	Calculates complete convolution for two arrays
2	Periodic convolution	ConvCyclic	Calculates periodic convolution for two arrays
3	Partial convolution	ConvPartial	Calculates partial convolution for two arrays
4	Correlation	Correlate	Calculates correlation for two arrays
5	Periodic correlation	CorrCyclic	Calculates periodic correlation for two arrays

When using these functions, allocate one of the two input arrays to X memory, and the other to Y memory. The output array can be allocated to either memory.

(2) Explanation of each function

(a) Complete convolution

Description:

- Format:

```
int ConvComplete  (short output[], const short ip_x[], const short ip_y[], long
                   x_size, long y_size, int res_shift)
```

- Parameters:

`output[]`	Output z
`ip_x[]`	Input x
`ip_y[]`	Input y
`x_size`	Size X of ip_x
`y_size`	Size Y of ip_y
`res_shift`	Right shift applied to each output.

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`x_size < 1` •`y_size < 1` •`res_shift < 0` •`res_shift > 25`

- Explanation of this function:

Complete convolves the two input arrays x and y, and writes the results to the output array z.

- Remarks:

$$z(m) = \left[\sum_{i=0}^{X-1} x(i)\, y(m-i)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < X+Y-1$$

Data external to the input array is read as 0.
ip_x is allocated to X memory, ip_y is allocated to Y memory, and output is allocated to arbitrary memory.
In addition, it is necessary to ensure that the array output size is more than (xsize+ysize-1).

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define NX    8
#define NY    8
#define NOUT  NX+NY-1
#pragma section X
static short datx[NX];
#pragma section Y
static short daty[NY];
#pragma section
short w1[5] = {-1, -32768, 32767, 2, -3, };
short x1[5] = {1, 32767, -32767, -32767, -2, };
void main()
{
    short i;
    short output[NOUT];
    int xsize, ysize, rshift;
    /* copy data into X and Y RAM */
    for(i=0;i<NX;i++){
        datx[i] = w1[i%5];
    }
    for(i=0;i<NY;i++){
        daty[i] = x1[i%5];
    }
    xsize = NX;
    ysize = NY;
    rshift = 15;
    if(ConvComplete(output, datx, daty, xsize, ysize, rshift) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<NX;i++){
        printf("#%3d  dat_x:%6d  dat_y:%6d ¥n",i,datx[i],daty[i]);
    }
    for(i=0;i<NOUT;i++){
        printf("#%3d  output:%d ¥n",i,output[i]);
    }
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data for use in convolution calculations.

(b) Periodic convolution

Description:

- Format:

```
int ConvCyclic  (short output[], const short ip_x[],
                 const short ip_y[], long size,
                 int res_shift)
```

- Parameters:

`output[]`	Output z
`ip_x[]`	Input x
`ip_y[]`	Input y
`size`	Size N of the array
`res_shift`	Right shift applied to each output.

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 1` •`res_shift < 0` •`res_shift > 25`

- Explanation of this function:

Periodically convolves the two input arrays x and y, and writes the results to the output array z.

- Remarks:

$$z(m) = \left[\sum_{i=0}^{N-1} x(i)\, y\left(|m - i + N|_N \right)\right] \cdot 2^{-res\_shift}$$
$$0 \le m < N$$

Here, $|i|_N$ means the remainder (i % N).
ip_x is allocated to X memory, ip_y is allocated to Y memory, and output is allocated to arbitrary memory.
In addition, it is necessary to ensure that the array output size is more than ‘size’.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
short x2[5] = {1, 32767, -32767, -32767, -2, };
short w2[5] = {-1, -32768, 32767, 2, -3, };
#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    short i;
    short output[N];
    int size, rshift;
    /* copy data into X and Y RAM */
    for(i=0;i<N;i++){
        datx[i] = w2[i];
        daty[i] = x2[i];
    }
    size = N ;
    rshift = 15;
    if(ConvCyclic(output, datx, daty, size, rshift) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }

    for(i=0;i<N;i++){
        printf("#%2d  ip_x:%6d  ip_y:%6d  output:%6d ¥n",
               i,datx[i],daty[i], output[i]);
    }
}
```

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data for use in convolution calculations.

(c) Partial convolution

Description:

- Format:

```
int ConvPartial  (short output[], const short ip_x[],
                  const short ip_y[], long x_size, long y_size, int res_shift)
```

- Parameters:

output[]	Output z
ip_x[]	Input x
ip_y[]	Input y
x_size	Size x of ip_x
y_size	Size y of ip_y
res_shift	Right shift applied to each output.

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	In any of the following cases •x_size < 1 •y_size < 1 •res_shift < 0 •res_shift > 25

- Explanation of this function:

This function convolves the two input arrays x and y, and writes the results to the output array z.

- Remarks:

Output fetched from data external to the input array is not included.

$$z(m) = \left[\sum_{i=0}^{A-1} a(i)\ b(m + A - 1 - i)\right] \cdot 2^{-res\_shift} \quad 0 \le m \le |A-B|$$

However, the number of arrays is a < b, and A is a size and B is b size.
Data external to the input array is read as 0.
ip_x is allocated to X memory, ip_y is allocated to Y memory, and output is allocated to arbitrary memory.
In addition, it is necessary to ensure that the array output size is more than (|xsize-ysize|+1).

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
```

Include header

```
#define    NX    5
#define    NY    5
short x3[5] = {1, 32767, -32767, -32767, -2, };
short w3[5] = {-1, -32768, 32767, 2, -3, };

#pragma section X
static short datx[NX];
#pragma section Y
static short daty[NY];
```

Variables placed in X or Y memory are defined by a pragma section within the section.

```
#pragma section
void main()
{
    short i;
    short output[NY+NX];
    int ysize, xsize, rshift;
    /* copy data into X and Y RAM */
    for(i=0;i<NX;i++){
        datx[i] = w3[i];
    }
    for(i=0;i<NY;i++){
        daty[i] = x3[i];
    }
```

Sets data for use in convolution calculations.

```
    xsize = NX;
    ysize = NY;
    rshift = 15;
    if(ConvPartial(output, datx, daty, xsize, ysize, rshift) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<NX;i++){
        printf("ip_x=%d  ¥n",datx[i]);
    }
    for(i=0;i<NY;i++){
        printf("ip_y=%d  ¥n",daty[i]);
    }
    for(i=0;i<(NY+NX);i++){
        printf("output=%d ¥n",output[i]);
    }
}
```

(d) Correlation

Description:

- Format:

```
int Correlate  (short output[], const short ip_x[],
                 const short ip_y[], long x_size, long y_size,
                 long no_corr, int x_is_larger, int res_shift)
```

- Parameters:

output[]	Output z
ip_x[]	Input x
ip_y[]	Input y
x_size	Size x of ip_x
y_size	Size y of ip_y
no_corr	Number of correlations M for calculation
x_is_larger	Array specification when x=y
res_shift	Right shift applied to each output.

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	In any of the following cases •x_size < 1 •y_size < 1 •no_corr < 1 •res_shift < 0 •res_shift > 25 •x_is_larger ≠ 0 or 1

- Explanation of this function:

Finds the correlation of the two input arrays x and y, and writes the results to the output array z.

- Remarks:

In the following equation, the number of arrays is a < b, and A is a size. If x_is_larger=0 make x to be a, and if x_is_larger=1 make x to be b.
Operation is not guaranteed when the b array is smaller than the a array.
Set the sizes of the input arrays x and y, as well as x_is_larger, so that no conflict exists.

$$z(m) = \left[\sum_{i=0}^{A-1} a(i)\, b(i+m)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < M$$

There is no obstacle to having A < X + M. In this case, use 0 for data external to the array.
res_shift=0 corresponds to normal integer calculation, and res_shift=15 corresponds to decimal calculation.
ip_x is allocated to X memory, ip_y is allocated to Y memory, and output is allocated to arbitrary memory.
In addition, it is necessary to ensure that the array output size is more than no_corr.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define NY   5
#define NX   5
#define M    4
#define MAXM  NX+NY
short x4[5] = {1, 32767, -32767, -32767, -2, };
short w4[5] = {-1, -32768, 32767, 2, -3, };
#pragma section X
static short datx[NX];
#pragma section Y
static short daty[NY];
#pragma section
void main()
{
    short i;
    int ysize, xsize, ncorr, rshift;
    short output[MAXM];
    int x_is_larger;
    /* copy data into X and Y RAM */
    for(i=0;i<NX;i++){
         datx[i] = w4[i%5];
    }
    for(i=0;i<NY;i++){
        daty[i] = x4[i%5];
    }
    /* test working of stack */
    ysize = NY;
    xsize = NX;
    ncorr = M;
    rshift = 15;
    x_is_larger=0;
    for (i = 0; i < MAXM; output[i++] = 0);
    if (Correlate(output, datx, daty, xsize, ysize, ncorr,
                                   x_is_larger,rshift) != EDSP_OK){
       printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<MAXM;i++){
        printf("[%d]:output=%d¥n",i,output[i]);
    }
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data for use in calculations.

(e) Periodic correlation

Description:

- Format:

```
int CorrCyclic  (short output[], const short ip_x[],
                        const short ip_y[], long size, int reverse,
                        int res_shift)
```

- Parameters:

output[]	Output z
ip_x[]	Input x
ip_y[]	Input y
size	Size N of the array
reverse	Reverse flag
res_shift	Right shift applied to each output.

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	In any of the following cases •size < 1 •res_shift < 0 •res_shift > 25 •reverse ≠ 0 or 1

- Explanation of this function:

Finds the correlation of the two input arrays x and y periodically, and writes the results to the output array z.

- Remarks:

$$z(m) = \left[\sum_{i=0}^{N-1} x(i)\, y\left(\left|i + m\right|_N\right)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < N$$

Here, $|i|_N$ means the remainder (i % N). If reverse=1, the output data is reversed, and the actual calculation is as follows.

$$z(m) = \left[\sum_{i=0}^{N-1} y(i)\, x\left(\left|i + m\right|_N\right)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < N$$

ip_x is allocated to X memory, ip_y is allocated to Y memory, and output is allocated to arbitrary memory.
In addition, it is necessary to ensure that the array output size is more than 'size'.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
```

Include header

```
#define N 5
short x5[5] = {1, 32767, -32767, -32767, -2, };
short w5[5] = {-1, -32768, 32767, 2, -3, };

#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
```

Variables placed in X or Y memory are defined by a pragma section within the section.

```
#pragma section
void main()
{
    short i;
    short output[N];
    int size, rshift;
    int reverse;
    int result;
    /* TEST CYCLIC CORRELATION OF X WITH Y */
    reverse=0;
    /* copy data into X and Y RAM */
    for(i=0;i<N;i++){
        datx[i] = w5[i];
        daty[i] = x5[i];
    }
```

Sets data for use in calculations.

```
    /* test working of stack */
    size = N;
    rshift = 15;
    if (CorrCyclic(output, datx, daty, size, reverse, rshift) != EDSP_OK){
        printf("EDSP_OK not returned - this one¥n");
    }
    for(i=0;i<N;i++){
        printf("output[%d]=%d¥n",i,output[i]);
    }
}
```

2.2.5 Other

(1) List of functions

Table 2.7 DSP Library Function List (Miscellaneous)

No.	Type	Function Name	Description
1	H’8000 → H’8001 replacement	Limit	Replaces H'8000 data with H'8001
2	X memory → Y memory copy	CopyXtoY	Copies an array from X memory to Y memory.
3	Y memory → X memory copy	CopyYtoX	Copies an array from Y memory to X memory.
4	Copy to X memory	CopyToX	Copies an array from a specified location to X memory.
5	Copy to Y memory	CopyToY	Copies an array from a specified location to Y memory.
6	Copy from X memory	CopyFromX	Copies an array from X memory to a specified location.
7	Copy from Y memory	CopyFromY	Copies an array from Y memory to a specified location.
8	Gaussian white noise	GenGWnoise	Generates Gaussian white noise.
9	Matrix multiplication	MatrixMult	Multiplies two matrices.
10	Multiplication	VectorMult	Multiplies two data elements.
11	RMS value	MsPower	Determines RMS power.
12	Mean	Mean	Determines mean.
13	Mean and variance	Variance	Determines mean and variance.
14	Maximum value	MaxI	Determines maximum value of integer array.
15	Minimum value	MinI	Determines minimum value of integer array.
16	Maximum absolute value	PeakI	Determines maximum absolute value of integer array.

(2) Explanation of each function

(a) H’8000 → H’8001 replacement

Description:

- Format:
  `int Limit (short data[], long no_elements, int data_is_x)`

- Parameters:

`data[]`	Data array
`no_elements`	Number of data elements
`data_is_x`	Data location specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases
	•`no_elements < 1`
	•`data_is_x ≠ 0 or 1`

- Explanation of this function:

Replaces input data with a value of H'8000 with H'8001. In this way, when fixed point multiplication is performed with the DSP instruction, overflow will not occur.

- Remarks:

Even when the process is performed there is a possibility that overflow may occur with addition in the multiply-and-accumulate operation.
When data_is_x=1 allocate data to X memory, and when data_is_x=0 allocate data to Y memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    short i;
    int size;
    int src_x;    /* copy data into X and Y RAM */
    for(i=0;i<N;i++) {
        datx[i] = dat[i%4];
        daty[i] = dat[i%4];
        printf("BEFORE NO %d datx daty :%d:%d ¥n",i,datx[i], daty[i]);
    }
    size = N;
    src_x = 1;

    if (Limit(datx, size, src_x) != EDSP_OK){
        printf( "EDSP_OK not returned¥n");
    }
    src_x = 0;
    if (Limit(daty, size, src_x) != EDSP_OK){
        printf( "EDSP_OK not returned¥n");
    }
    for(i=0;i<N;i++) {
        printf("After NO %d datx daty :%d:%d¥n",i,datx[i], daty[i]);
    }
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

If using X memory

If using Y memory

(b) X memory → Y memory copy

Description:

- Format:
  `int CopyXtoY  (short op_y[], const short ip_x[], long n)`

- Parameters:

`op_y[]`	Output array
`ip_x[]`	Input array
`n`	Number of data elements

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	`n < 1`

- Explanation of this function:

The array is copied from ip_x to op_y.

- Remarks:

Allocate ip_x to X memory, and allocate op_y to Y memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    int i;

    for(i=0;i<N;i++){
        daty[i]=0;
        datx[i]=dat[i%4];
    }
    if(CopyXtoY(daty, datx, N) != EDSP_OK){
        printf("CopyXtoY Problem¥n");
    }
    printf("no_elements:%d ¥n",N);
    for(i=0;i<N;i++){
        printf("#%2d  op_x:%6d  ip_y:%6d ¥n",i,datx[i],daty[i]);
    }
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

(c) Y memory → X memory copy

Description:

- Format:
  `int CopyYtoX (short op_x[], const short ip_y[], long n)`

- Parameters:

op_x[]	Output array
ip_y[]	Input array
n	Number of data elements

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	n < 1

- Explanation of this function:

The array is copied from ip_y to op_x.

- Remarks:

Allocate ip_y to Y memory, and allocate op_x to X memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  5
static short dat[N] = { -32768, 32767, -32768, 0,3};
#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    int i;

    for(i=0;i<N;i++){
        daty[i]=dat[i];
    }
    if(CopyYtoX(datx, daty, N)!= EDSP_OK){
     printf("CopyYtoX error!¥n");
    }
    printf("no_elements %d ¥n",N);
    for(i=0;i<N;i++){
        printf("#%2d  po_x:%6d  ip_y:%6d ¥n",i,datx[i],daty[i]);
    }
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

(d) Copy to X memory

Description:

- Format:
  `int CopyToX (short op_x[], const short input[], long n)`

- Parameters:

op_x[]	Output array
input[]	Input array
n	Number of data elements

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	n < 1

- Explanation of this function:

The array input is copied to op_x.

- Remarks:

Allocate op_x to X memory, and allocate input to arbitrary memory.

Example of use:

```
#include <stdio.h>        Include header
#include <ensigdsp.h>
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
#pragma section X
static short datx[N];     Variables placed in X memory are defined by a pragma section within the section.
#pragma section
void main()
{
    int   i;
    short data[N];

    for(i=0;i<N;i++){     Sets data.
       data[i]=dat[i];
    }
    if(CopyToX(datx, data, N) !=EDSP_OK){
        printf("CopyToX Problem¥n");
    }
    printf("no_elements %d¥n",N);
    for(i=0;i<N;i++){
        printf("#%2d  op_x:%6d  input:%6d ¥n",i,datx[i],data[i]);
    }
}
```

(e) Copy to Y memory

Description:

- Format:
  `int CopyToY (short op_y[], const short input[], long n)`

- Parameters:

`op_y[]`	Output array
`input[]`	Input array
`n`	Number of data elements

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	`n < 1`

- Explanation of this function:

The array input is copied to op_y.

- Remarks:

Allocate op_y to Y memory, and allocate input to arbitrary memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    int i;
    short data[N] ;

    for(i = 0; i < N; i++){
        data[i] = dat[i%4] ;
    }
    if(CopyToY(daty, data, N) != EDSP_OK){
        printf("CopyToY Problem¥n");
    }
    printf("no_elements %ld ¥n",N);
    for(i = 0; i < N; i++){
        printf("#%2d  op_y:%6d  input:%6d ¥n",i,daty[i],data[i]);
    }
}
```

Include header

Variables placed in Y memory are defined by a pragma section within the section.

Sets data.

(f) Copy from X memory

Description:

- Format:
  `int CopyFromX (short output[], const short ip_x[], long n)`

- Parameters:

`output[]`	Output array
`ip_x[]`	Input array
`n`	Number of data elements

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	`n < 1`

- Explanation of this function:

The array ip_x is copied to output.

- Remarks:

Allocate ip_x to X memory, and allocate output to arbitrary memory.

Example of use:

Include header

Variables placed in X memory are defined by a pragma section within the section.

Sets data.

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
static short out_dat[N] ;

#pragma section X
static short datx[N];
#pragma section
void main(){
    int i;

    for(i=0;i<N;i++){
        datx[i]=dat[i];
    }
    if(CopyFromX(out_dat,datx, N) != EDSP_OK){
        printf("CopyFromX Problem¥n");
    }
    for(i=0;i<N;i++){
        printf("#%3d output:%6d  ip_x:%6d ¥n",i,out_dat[i],datx[i]);
    }
    printf("no_elements:%ld¥n",N);
}
```

(g) Copy from Y memory

Description:

- Format:
  `int CopyFromY  (short output[], const short ip_y[], long n)`

- Parameters:

output[]	Output array
ip_y[]	Input array
n	Number of data elements

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	n < 1

- Explanation of this function:

The array ip_y is copied to output.

- Remarks:

Allocate ip_y to Y memory, and allocate output to arbitrary memory.

Example of use:

```
#include <stdio.h>        } Include header
#include <ensigdsp.h>
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
static short out_dat[N] ;
                                    Variables placed in Y memory are
#pragma section Y                   defined by a pragma section within the
static short daty[N];               section.
#pragma section
void main()
{
    int i;
                                    Sets data.
    for(i=0;i<N;i++){
        daty[i]=dat[i];
    }
    if(CopyFromY(out_dat,daty, N)!= EDSP_OK){
         printf("CopyFormY Problem¥n");
    }
    printf("no_elements:%d ¥n",N);
    for(i=0;i<N;i++){
        printf("#%2d  output:%6d  ip_y:%6d ¥n",i,out_dat[i],daty[i]);
    }
}
```

(h) Gaussian white noise

Description:

- Format:
`int GenGWnoise (short output[], long no_samples, float variance)`

- Parameters:

`output[]`	Outputs white noise data
`no_samples`	Number of output data elements
`Variance`	Variance of noise distribution σ^2

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_samples < 1` •`variance ≤ 0.0`

- Explanation of this function:

With a mean of 0, Gaussian white noise is generated with the variance specified by the user.

- Remarks:

One set of two output data elements are generated. In order to generate 1 set of output data, use a rand function, and until a set of less than 1 is found by the sum of the square of x, 1 set of random numbers, γ_1 and γ_2, between –1 and 1 is generated. Then 1 set of output data, o_1 and o_2, is calculated using the following equations.

$$o_1 = \sigma\gamma_1\sqrt{-2\ln(x)/x}$$

$$o_2 = \sigma\gamma_2\sqrt{-2\ln(x)/x}$$

If the number of data elements is set to an odd number, the second data element of the last set is nullified.
As the rand function of the standard library called on by this function is not reentrant, the order of the random numbers γ_1 and γ_2 generated will not necessarily always be the same. However, there will be no impact on the characteristics of the white noise o_1 and o_2 generated.
This function uses a floating point operation. As the processing speed of floating point operations is slow, it is recommended that this function is used for evaluation.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>       } Include header
#define MAXG   4.5        /* approx. saturating level for N(0,1) random variable */
#define N_SAMP  10        /* number of samples generated in a frame        */
void main()
{
    short out[N_SAMP];
    float var;
    int   i;
    var = 32768 / MAXG * 32768 / MAXG;
    if(GenGWnoise(out, N_SAMP, var) !=EDSP_OK){
        printf("GenGWnoise Problem¥n");
    }
    for(i=0;i<N_SAMP;i++){
        printf("#%2d  out:%6d  ¥n",i,out[i]);
    }
}
```

(i) Matrix multiplication

Description:

- Format:

```
int MatrixMult  (void *op_matrix, const void *ip_x,
                 const void *ip_y, long m, long n, long p,
                 int x_first, int res_shift)
```

- Parameters:

`op_matrix`	Pointer to the first data element of output
`ip_x`	Pointer to the first data element of input x
`ip_y`	Pointer to the first data element of input y
`m`	Number of rows in matrix 1
`n`	Number of columns in matrix 1, number of rows in matrix 2
`p`	Number of rows in matrix 2
`x_first`	Order specification for matrix multiplication
`res_shift`	Right shift applied to each output.

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`m, n,` or `p < 1` •`res_shift < 0` •`res_shift > 25` •`x_first ≠ 0` or `1`

- Explanation of this function:

Performs multiplication of the two matrices x and y, and allocates the result to op_matrix.

- Remarks:

When x_first=1, calculates x · y In this case, ip_x is m x n, ip_y is n x p, and op_matrix is m x p.
When x_first=0, calculates y · x. In this case, ip_y is m x n, ip_x is n x p, and op_maxtrix is m x p.
The results of multiply-and-accumulate operations are saved as 39 bits. Output y(n) is the lower 16 bits fetched from the res_shift bit right shifted results. When an overflow occurs, this is the positive or negative maximum value.
Each matrix is allocated to a normal C format (row major order).

a_0	a_1	a_2	a_3
a_4	a_5	a_6	a_7
a_8	a_9	a_{10}	a_{11}

In order to be able to specify an arbitrary array size, specify void* for the array parameters. Make these parameters point to short variables.
Provide input arrays ip_x and ip_y, and output array op_matrix separately.
Allocate ip_x to X memory, allocate ip_y to Y memory, and allocate op_matrix to arbitrary memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  4
#define NN  N*N
short m1[16] = {  1, 32767, -32767, 32767,
                  1, 32767, -32767, 32767,
                  1, 32767, -32767, 32767,
                  1, 32767, -32767, 32767, };
short m2[16] = {  -1, 32767, -32767, -32767,
                  -1, 32767, -32767, -32767,
                  -1, 32767, -32767, -32767,
                  -1, 32767, -32767, -32767, };
#pragma section X
static short datx[NN];
#pragma section Y
static short daty[NN];
#pragma section
void main()
{
    short i, j;
    short output[NN];
    int m, n, p, rshift, x_first;
    long sum;
    for (i = 0; i < NN; output[i++] = 0) ;
    /* copy data into X and Y RAM */
    for(i=0;i<NN;i++) {
        datx[i] = m1[i%16];
        daty[i] = m2[i%16];
    }
    m = n = p = N;
    rshift = 15;
    x_first = 1;
    if (MatrixMult(output, datx, daty, m, n, p, x_first, rshift) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<NN;i++) {
        printf("output[%d]=%d¥n",i,output[i]);
    }
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

(j) Multiplication

Description:

- Format:

```
int VectorMult  (short output[], const short ip_x[],
                 const short ip_y[],long no_elements, int res_shift)
```

- Parameters:

`output[]`	Output
`ip_x[]`	Input 1
`ip_y[]`	Input 2
`no_elements`	Number of data elements
`res_shift`	Right shift applied to each output.

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_elements < 1` •`res_shift < 0` •`res_shift > 16`

- Explanation of this function:

Data is fetched one element at a time from ip_x and ip_y and multiplication is performed, with the results being allocated to output.

- Remarks:

Output is the lower 16 bits fetched from the res_shift bit right shifted results.
When an overflow occurs, this is the positive or negative maximum value.
This function performs multiplication of the data. To calculate the inner product, use the MatrixMult function, setting 1 for m (the number of rows of matrix 1) and for p (the number of columns of matrix 2).
ip_x is allocated to X memory, ip_y is allocated to Y memory, and output is allocated to arbitrary memory.

Include header

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  4
#define RSHIFT  15
short y[4] = {1, 32767, -32767, 32767, };
short x[4] = {-1, 32767, -32767, -32767, };

#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    short i, n ;
    short output[N];
    int size, rshift;
    /* copy data into X and Y RAM */
    for(i=0;i<N;i++) {
        datx[i] = x[i];
        daty[i] = y[i];
    }
    size = N;
    rshift = RSHIFT;
    for (i = 0; i < N; output[i++] = 0) ;
    if (VectorMult(output, datx, daty, size, rshift) != EDSP_OK) {
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<N;i++){
        printf("#%2d  output:%6d  ip_x:%6d  ip_y:%6d ¥n",i,
                                          output[i],datx[i], daty[i]);
    }
}
```

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

(k) RMS value

Description:

- Format:

```
int MsPower  (long *output, const short input[],long no_elements, int src_is_x)
```

- Parameters:

`output`	Pointer to output
`input[]`	Input x
`no_elements`	Number of data elements N
`src_is_x`	Data location specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_elements < 1` •`src_is_x ≠ 0 or 1`

- Explanation of this function:

Determines the RMS value of input data.

- Remarks:

$$RMS = \frac{1}{N}\sum_{i=0}^{N-1} x(i)^2$$

Rounds off the division result to the nearest integer.
The result of the operation is saved as 63 bits.
If no_elements is 2^{32}, overflow may occur.
When src_is_x=1 allocate input to X memory, and when src_is_x=0 allocate data to Y memory.
Allocate output to arbitrary memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
```

Include header

```
#define N  5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};

#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
```

Variables placed in X or Y memory are defined by a pragma section within the section.

```
#pragma section
void main()
{
    int     i;
    long    output[1];
    int     src_x;

    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
```

Sets data.

When X memory is used, src_x=1.

```
    src_x = 1;
    if (MsPower(output, datx, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("MsPower:x=%d¥n",output[0]);
    src_x = 0;
    if (MsPower(output, daty, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("MsPower:y=%d¥n",output[0]);
}
```

When Y memory is used, src_x=0.

(l) Mean

Description:

- Format:

```
int Mean  (short *mean, const short input[], long no_elements, int src_is_x)
```

- Parameters:

`mean`	Pointer to mean value of input
`input[]`	Input x
`no_elements`	Number of data elements N
`src_is_x`	Data location specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_elements < 1` •`src_is_x` ≠ 0 or 1

- Explanation of this function:

Determines the mean of input data.

- Remarks:

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

Rounds off the division result to the nearest integer.
The operation result is saved as 32 bits. If no_elements is greater than 2^{16}-1, overflow may occur.
When src_is_x=1 allocate input to X memory, and when src_is_x=0 allocate data to Y memory.

Example of use:

```
#include <stdio.h>        } Include header
#include <ensigdsp.h>
#define N  5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};
#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    short      i,output[1];
    int        size;
    int        src_x;
    int        flag = 1;
    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    /* test working of stack */
    src_x = 1;
    if (Mean(output, datx, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Mean:x=%d¥n",output[0]);
    src_x = 0;
    if (Mean(output, daty, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Mean:y=%d¥n",output[0]);
}
```

Variables placed in X or Y memory are defined by a pragma section within the section.

When X memory is used, src_x=1.

When Y memory is used, src_x=0.

(m) Mean and variance and

Description:

- Format:

```
int Variance  (long *variance, short *mean, const short input[],
               long no_elements, int src_is_x)
```

- Parameters:

`Variance`	Pointer to the variance σ^2 of input
`mean`	Pointer to data mean $\bar{x}$
`input[]`	Input x
`no_elements`	Number of data elements N
`src_is_x`	Data location specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_elements < 1` •`src_is_x` ≠ 0 or 1

- Explanation of this function:

Determines mean and variance of input.

- Remarks:

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

$$\sigma^2 = \frac{1}{N}\sum_{i=0}^{N-1} x(i)^2 - \bar{x}^2$$

Rounds off the division result to the nearest integer.
x is saved as 32 bits. There is no check for overflow.
If no_elements is greater than 2^{16}-1, overflow may occur.
σ^2 is saved as 63 bits. There is no check for overflow.
When src_is_x=1 allocate input to X memory, and when src_is_x=0 allocate data to Y memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};
#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    long        size,var[1];
    short       mean[1];
    int         i ;
    int         src_x;

    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }

    /* test working of stack */
    size = N;
    src_x = 1;
    if (Variance(var, mean, datx, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Variance:%d  mean:%d ¥n ",var[0],mean[0]);
    src_x = 0;
    if (Variance(var, mean, daty, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Variance:%d  mean:%d ¥n ",var[0],mean[0]);
}
```

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

When X memory is used, src_x=1.

When Y memory is used, src_x=0.

(n) Maximum value

Description:

- Format:

```
int MaxI  (short **max_ptr, short input[], long no_elements, int src_is_x)
```

- Parameters:

`max_ptr`	Pointer to the maximum data
`input[]`	Input
`no_elements`	Number of data elements
`src_is_x`	Data location specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_elements < 1` •`src_is_x ≠ 0 or 1`

- Explanation of this function:

Searches for the maximum value in the array input, and returns its address to max_ptr.

- Remarks:

If several data elements have the same maximum value, the address of the data with the start closest to input is returned. When src_is_x=1 allocate input to X memory, and when src_is_x=0 allocate data to Y memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
static short dat[131] = {-16384, -32767, 32767, 14877, 8005};
#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    short    *outp,**outpp;
    int       size,i;
    int       src_x;
 /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    /* MAXI */
    size = N;
    outpp = &outp;
    src_x = 1;
    if (MaxI(outpp, datx, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Max:x = %d¥n",**outpp);
    src_x = 0;
    if (MaxI(outpp, daty, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Max:y = %d¥n",**outpp);
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

When X memory is used, src_x=1.

When Y memory is used, src_x=0.

(o) Minimum value

Description:

- Format:

```
int MinI  (short **min_ptr, short input[], long no_elements, int src_is_x)
```

- Parameters:

`min_ptr`	Pointer to the minimum data
`input[]`	Input
`no_elements`	Number of data elements
`src_is_x`	Data location specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_elements < 1` •`src_is_x ≠ 0 or 1`

- Explanation of this function:

Searches for the minimum value in the array input, and returns its address to min_ptr.

- Remarks:

If several data elements have the same minimum value, the address of the data with the start closest to input is returned. When src_is_x=1 allocate input to X memory, and when src_is_x=0 allocate data to Y memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 10
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};
#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    short     *outp,**outpp;
    int       size,i;
    int       src_x;
    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    /* MINI */
    /* test working of stack */
    size = N;
    outpp = &outp;
    src_x = 1;
    if (MinI(outpp, datx, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Min:x=%d¥n",**outpp);
    src_x = 0;
    if (MinI(outpp, daty, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Min:y=%d¥n",**outpp);
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

When X memory is used, src_x=1.

When Y memory is used, src_x=0.

(p) Maximum absolute value

Description:

- Format:

```
int PeakI  (short **peak_ptr, short input[], long no_elements, int src_is_x)
```

- Parameters:

`peak_ptr`	Pointer to the maximum absolute value data
`input[]`	Input
`no_elements`	Number of data elements
`src_is_x`	Data location specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_elements < 1` •`src_is_x ≠ 0 or 1`

- Explanation of this function:

Searches for the maximum absolute value in the array input, and returns its address to peak_ptr.

- Remarks:

If several data elements have the same maximum absolute value, the address of the data with the start closest to input is returned.
When src_is_x=1 allocate input to X memory, and when src_is_x=0 allocate data to Y memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};
#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    short     *outp,**outpp;
    int       size,i;
    int       src_x;
    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    size = N;
    outpp = &outp;
    src_x = 1;
    if (PeakI(outpp, datx, size, src_x) != EDSP_OK)
    {
        printf("EDSP_OK not returned¥n");
    }
    printf("Peak:x=%d¥n",**outpp);
    src_x = 0;
    if (PeakI(outpp, daty, size, src_x) != EDSP_OK)
    {
        printf("EDSP_OK not returned¥n");
    }
    printf("Peak:y=%d¥n",**outpp);
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

When X memory is used, src_x=1.

When Y memory is used, src_x=0.

2.3 Performance of the DSP Library

(1) Number of execution cycles of the DSP library

The number of execution cycles required by functions in the DSP library are indicated below. Measurements were performed using an emulator (SH-DSP, 60 MHz), with the program section allocated to X-ROM or to Y-ROM.

Table 2.8 List of Execution Cycles for DSP Library Functions (1)

Category	DSP Library Function Name	Number of Execution Cycles (Cycle)	Notes
Fast Fourier transforms	FftComplex	29,330	Size: 256 Scaling: 0xFFFFFFFF
	FftReal	25,490	
	IfftComplex	30,380	
	IfftReal	29,240	
	FftInComplex	26,540	
	FftInreal	25,260	
	IfftInComplex	27,590	
	IfftInReal	27,470	
	LogMagnitude	1,778,290	
	InitFft	3,116,640	
	FreeFft	780	
Filter functions	Fir	23,010	Number of coefficients: 64 Number of data items: 200 Convergence coefficient 2μ = 32767
	Fir1	280	
	Lms	97,710	
	Lms1	790	
	InitFir	1,400	
	InitLms	1,400	
	FreeFir	90	
	FreeLms	90	
	Iir	23,530	Number of data items: 200 Number of filter sections: 5
	Iir1	360	
	DIir	309,010	
	DIir1	1,860	
	InitIir	280	
	InitDIir	280	
	FreeIir	90	
	FreeDIir	270	

Table 2.8 List of Execution Cycles for DSP Library Functions (2)

Category	DSP Library Function Name	Number of Execution Cycles (Cycle)	Notes
Window functions	GenBlackman	789,950	Number of data items: 100
	GenHamming	418,330	
	GenHanning	447,250	
	GenTriangle	744,220	
Convolution functions	ConvComplete	21,890	Number of data items: 100
	ConvCyclic	14,790	
	ConvPartial	370	
	Correlate	11,930	
	CorrCylic	15,790	
Other functions	Limit	480	Number of data items: 100
	CopyXtoY	130	
	CopyYtoX	130	
	CopyToX	1,270	
	CopyToY	1,270	
	CopyFromX	1,320	
	CopyFromY	1,320	
	GenGWnoise	2,878,410	
	MatrixMult	2,337,460	
	VectorMult	1,500	
	MsPower	370	
	Mean	270	
	Variance	820	
	MaxI	540	
	MinI	520	
	PeakI	740	

(2) Comparison of C language and DSP library source code

Here source code is presented in C language and from the DSP library, for some of the FFT-related functions (those performing butterfly calculations).

In the DSP library source code, the DSP-specific instructions such as movx, movy, and padd are used to improve the performance of the DSP library.

C source code

```
void R4add(short *arp, short *brp, short *aip, short *bip, int grpinc, int numgrp)
{
short  tr,ti;
int    grpind;
      for(grpind=0;grpind<numgrp;grpind++) {
               tr  =  *brp;
               ti  =  *bip;
               *brp = sub(*arp,ti);
               *bip = add(*aip,tr);
               *arp = add(*arp,ti);
               *aip = sub(*aip,tr);
               arp += grpinc;
               aip += grpinc;
               brp += grpinc;
               bip += grpinc;
      }
}
```

DSP library source code

```
_R4add:
        MOV.L Ix,@-R15
        MOV.L Iy,@-R15

        MOV.L @(2*4,R15),Ix
        SHLL  Ix
        MOV   Ix,Iy
        MOV.L @(3*4,R15),R1

        REPEAT r4alps,r4alpe
        ADD   #-1,R1
        SETRC R1
                                         movx.w  @ar,X0          movy.w @bi,Y0
        padd  X0,Y0,A0
        psub  X0,Y0,A1                   movx.w  @br,X0          movy.w @ai,Y0
        padd  X0,Y0,A0                   movx.w  A0,@ar+Ix
                          pneg    X0,X0  movx.w  A1,@br+Ix
        padd  X0,Y0,A1                                           movy.w A0,@bi+Iy
                                         movx.w  @ar,X0          movy.w @bi,Y0

        .ALIGN      4
r4alps  padd  X0,Y0,A0                                           movy.w A1,@ai+Iy
        psub  X0,Y0,A1                   movx.w  @br,X0          movy.w @ai,Y0
        padd  X0,Y0,A0                   movx.w  A0,@ar+Ix
                          pneg   X0,X0   movx.w  A1,@br+Ix
        padd  X0,Y0,A1                                           movy.w A0,@bi+Iy
r4alpe                                   movx.w  @ar,X0          movy.w @bi,Y0
                                                                 movy.w A1,@ai+Iy
        MOV.L @R15+,Iy
        RTS
        MOV.L @R15+,Ix
```

(3) Performance of individual FFT functions

Fourier transform functions are classified as follows.

Table 2.9 Fast Fourier Transform Functions

	Not-in-place function	In-place function
Complex Fourier transform	FftComplex	FftInComplex
Real Fourier transform	FftReal	FftReal

Table 2.10 Inverse Fast Fourier Transform Functions

	Not-in-place function	In-place function
Complex Fourier transform	IfftComplex	IfftInComplex
Real Fourier transform	IfftReal	IfftInReal

(a) Differences between In-Place and Not-In-Place Functions

In-place functions use the array of input data as the array for output data. Hence the input data is overwritten by the output data, and is not saved.

When using not-in-place functions, the input and output data must be prepared separately before calling on a function. The input data and output data are separate, and so the input data is saved even after the function is called on.

There is almost no difference in the performance of in-place and not-in-place functions, and so the type of function to be used should be determined based on the amount of memory available.

Compared with not-in-place functions, in-place functions require half the amount of memory.

- About scaling

In each stage of FFT calculations, calculations are executed in multiply-and-accumulate form, so overflows tend to occur. If an overflow occurs, all values become maxima or minima, so that calculation results cannot be evaluated correctly.

In order to prevent overflow, scaling is performed at each stage of FFT calculations; the scaling is 2 by which values are divided (right-shifted).

Table 2.11 Scaling Values and Features

Scaling Value	Features
FFTNOSCALE	No shifting whatsoever; overflow tends to occur
EFFTMIDSCALE	Shifting at every other stage
EFFTALLSCALE	Shifting at all stages; overflow does not occur readily

Scaling does not have a large effect on performance. Hence when deciding on a scaling, the features of the data, rather than performance, should be considered.

(4) Filter functions

(b) Using Fir and Lms

The relation between the number of coefficients and cycles for the Fir and Lms filters are shown in figure 3.11.

Because the Lms filter uses an adaptive algorithm, speed of calculation is slower than for the Fir filter. In a system with stable data waveforms, Lms should be used to determine filter coefficients, after which it should be replaced by the Fir filter.

The number of right-shifts can be specified for data scaling. Because multiply-and-accumulate operations are used internally in SH-DSP library functions, depending on the input data, overflows may occur. In such cases the number of right-shifts should be modified appropriately, and should be selected referring to output values.

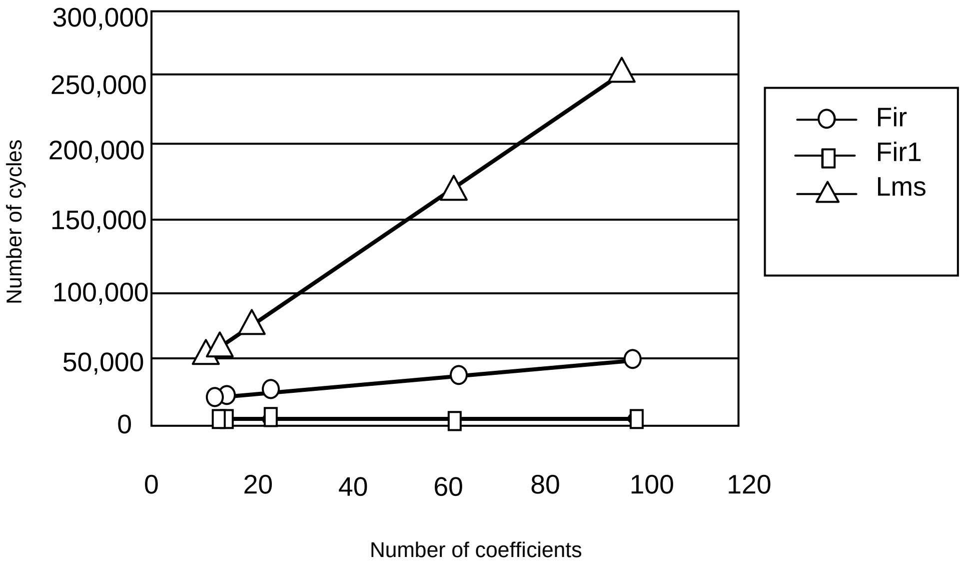


Figure 2.2 Relation between Number of Coefficients and Number of Cycles

- Iir and DIir

When performance is given priority, Iir should be used instead of DIir. Because multiply-and-accumulate operations are used internally in SH-DSP library functions, depending on the input data, overflows may occur. In such cases the number of right-shifts should be modified appropriately, and should be selected referring to output values.
The number of right-shifts can be specified for data scaling. However, the number of right-shifts is specified as part of the array of filter coefficients. For details, refer to section 3.13.6, (5)(c) IIR and (e) Double precision IIR.

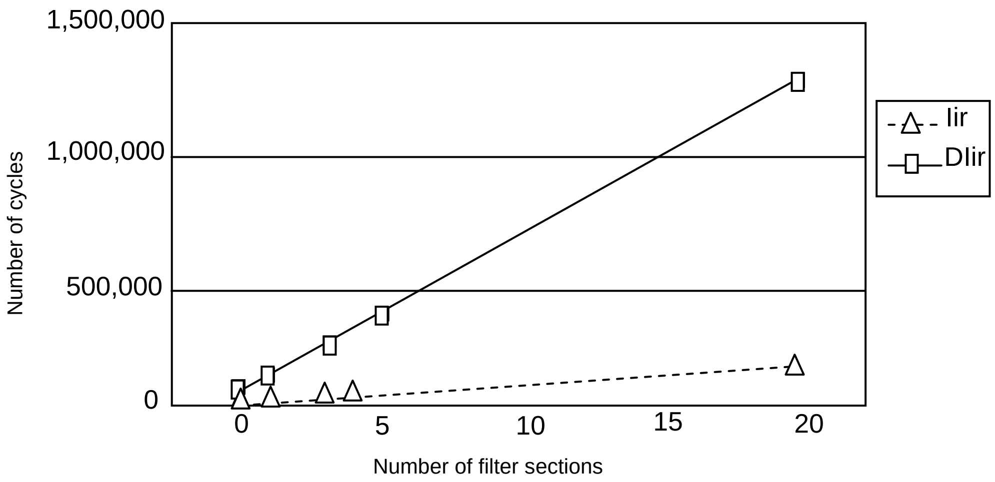


Figure 2.3 Relation between Number of Filter Sections and Number of Cycles

- Selective Use of Filter Functions

The Fir filter has a linear phase response and is always stable, making it suitable for use in audio, video and other applications where phase distortion cannot be tolerated. On the other hand, the Iir filter includes feedback, and can obtain results using fewer coefficients than Fir, for faster execution; it is suitable for applications where time constraints are imposed. However, the Iir filter may be unstable in some situations, and proper care should be taken in its use.

3. DSP-C Specifications

Description:

The DSP-C language is supported.
This specification is valid when the compiler option "dspc" is specified for the SuperH RISC engine C/C++ compiler.

3.1 Fixed-Point Data Type

Previously, the integer type has been used to represent a fractional value. You can now use the fixed-point data type to code a fractional value without modification.
The SuperH RISC engine C/C++ compiler generates DSP instructions appropriate to the fixed-point data type being used. Table 3.45 shows the internal representation of the fixed-point data type.

•

Table 3.1 Internal Representation of the Fixed-point Data Type

Type	Size (Size on memory)	Alignment number (bytes)	Range of data		Constant index
			Min. value	Max. value	
_ _fixed	16 bits (16 bits)	2	-1.0	$1.0\text{-}2^{-15}$ (0.999969482421875)	r
long _ _fixed	32 bits (32 bits)	4	-1.0	$1.0\text{-}2^{-31}$ (0.9999999995343387126922607421875)	R
_ _accum	24 bits (32 bits)	4	-256.0	$256.0\text{-}2^{-15}$ (255.999969482421875)	a
long _ _accum	40 bits (64 bits)	4	-256.0	$256.0\text{-}2^{-31}$ (255.9999999995343387126922607421875)	A

Important Information:

(1) The _ _accum and long _ _accum data stored in memory is right justified, with sign extension added at the beginning part.

Example: `(_ _accum)128.5a is stored as "00 40 40 00".`
Example: `(long _ _accum)(-256.0A) is stored as "FF FF FF 80 00 00 00 00".`

(2) Comparing DSP-C and the previous method

C function [Previous method]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8
short input[NUM] = {0x1000, 0x2000, 0x4000,
                    0x6000, 0xf000, 0xe000,
                    0xc000, 0xa000};
short result[NUM];
void func(void)
{
    int i;
    for (i = 0; i < NUM; i++) {
        result[i] = input[i] + 0x1000;
    }
}
void main(void)
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
        printf("%f¥n", result[i]/32768.0);
    }
}
```

[DSP-C]

```
// -cpu=sh3dsp -dspc
#include <stdio.h>
#define NUM 8
__fixed input[8] = { 0.125r, 0.25r, 0.5r, 0.75r,
                    -0.125r,  -0.25r, -0.5r,
                    -0.75r};
__fixed result[NUM];
void func()
{
    int i;
    for (i = 0; i < NUM; i++) {
        result[i] = input[i] + 0.125r;
    }
}
void main(void)
{
     int i;
    func();
    for (i = 0; i < NUM; i++) {
        printf("%r¥n", result[i]);
    }
}
```

(3) Example of multiply-and-accumulation operations

If the integer type is used as a substitute for a fractional value, the products must be aligned to the fixed number of digits. This alignment is unnecessary for the fixed-point data type.

C function [Previous method]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8
short x_input[NUM] = {0x1000, 0x2000, 0x4000,
                      0x6000, 0xf000, 0xe000,
                      0xc000, 0xa000};
short y_input[NUM] = {0x1000, 0x2000, 0x4000,
                      0x6000, 0xf000, 0xe000,
                      0xc000, 0xa000};
int result;
int func(short *x_input, short *y_input)
{
    int i;
    int temp = 0;
    for (i = 0; i< NUM ;i++) {
           temp += (x_input[i] * y_input[i]) >> 15;
    }
    return (temp);
}
void main()
{
    result = func(x_input, y_input);
    printf("%f¥n", result/32768.0);
}
```

[DSP-C]

```
// -cpu=sh3dsp -dspc -fixed_noround
#include <stdio.h>
#define NUM 8
_X fixed x_input[NUM] = { 0.125r,   0.25r,
                           0.5r,  0.75r,
                          -0.125r,  -0.25r,
                          -0.5r, -0.75r};
_Y_fixed y_input[NUM] = { 0.125r,   0.25r,
                          0.5r,  0.75r,
                          -0.125r,  -0.25r,
                          -0.5r, -0.75r};

__accum result;
void func(__accum *result_p,
          __X __fixed *x_input,
          __Y __fixed *y_input)
{
    int i;
    __accum temp = 0.0a;
    for (i = 0; i< NUM ;i++) {
        temp += x_input[i] * y_input[i];
    }
    *result_p = temp;
}
void main()
{
    func(&result, x_input, y_input);
    printf("%a¥n", result);
}
```

3.2 Memory Qualifier

Adding the X/Y memory qualifier to variables promotes generation of X/Y memory-dedicated access instructions which are more efficient than ordinary memory access instructions.
Use the following qualifier to explicitly specify the X or Y memory for storing data.

_ _X: Store data in the X memory.
_ _Y: Store data in the Y memory.

The SuperH RISC engine C/C++ compiler outputs objects that have the _ _X or _ _Y memory qualifier to the sections shown in table 3.46. You must allocate these sections to the X or Y memory during linking.

•

Table 3.2 Memory Qualifier Specifications

Name	Section	Description
Constant area	$XC	const data (Stored in the X memory)
	$YC	const data (Stored in the X memory)
Initialized data area	$XD	Data with an initial value (Stored in the X memory)
	$YD	Data with an initial value (Stored in the Y memory)
Uninitialized data area	$XB	Data without an initial value (Stored in the X memory)
	$YB	Data without an initial value (Stored in the Y memory)

However, X or Y memory may exist only on RAM. You must be careful when creating ROM from such memory.

Examples of use:

(1) Storing data in memory by using the _ _X or _ _Y memory qualifier

```
_ _X  int           a;      //Store in the X memory.
      int  _ _X     b;      //Store in the X memory.
_ _Y  int         * c;      //Pointer to the int data in the Y memory (Memory is undefined.)
      int   _ _Y  * d;      //Pointer to the int data in the Y memory (Memory is undefined.)
      int   *_ _Y   e;      //Pointer to the int data (Stored in the Y memory)
_ _X  int   *_ _Y   f;      //Pointer to the int data in the X memory (Stored in the Y memory)
```

(2) Copying the constant area and initialized data area from ROM to X/Y RAM
In this example, the data that was stored in ROM during linking is copied to X/Y RAM when the program starts. You need to use the VOW option of the optimizing linkage editor to allocate the same space twice in ROM and in X/Y RAM.

Example of the subcommand during linking:

```
rom=$XC=XC,$XD=XD,$YC=YC,$YD=YD
start    P,C,D,$XC,$XD,$YC,$YD/400,$XB,XC,XD/05007000,$YB,YC,YD/05017000
```

The standard library function INITSCT() allows you to easily copy data from ROM to X/Y RAM.

Example of use: _INITSCT()

```
#include <_h_c_lib.h>
void PowerON_Reset(void)
{
        _INITSCT();
    main();
     sleep();
}

#pragma section $DSEC
static const struct {
    void *rom_s;
    void *rom_e;
    void *ram_s;
} DTBL[] = { {__sectop("$XC"), __secend("$XC"), __sectop("XC")},
         {__sectop("$XD"), __secend("$XD"), __sectop("XD")},
         {__sectop("$YC"), __secend("$YC"), __sectop("YC")},
         {__sectop("$YD"), __secend("$YD"), __sectop("YD")}};
#pragma section
```

(3) Not using the constant area or initialized area

By specifying that neither a const specification nor initialized data is to be added to an object with the X/Y memory qualifier, you do not have to allocate the same space twice in ROM and in X/Y RAM.

For example, you can eliminate initialized data by specifying dynamic initialization as shown in the following example.

Example of use

```
#define NUM 8
__X __fixed x_input[NUM];
__Y __fixed y_input[NUM];
__fixed x_input[NUM] = { 0.125r,  0.25r, 0.5r, 0.75r, -0.125r, -0.25r, -0.5r, -0.75r};
__fixed y_input[NUM] = { 0.125r,  0.25r, 0.5r, 0.75r, -0.125r, -0.25r, -0.5r, -0.75r};
void xy_init()
{
      int i;

      for (i = 0; i< NUM; i++) {
          x_input[i] = x_init[i];
          y_input[i] = y_init[i];
      }
}

void main()
{
       xy_init();
          :
}
```

(4) Comparing DSP-C and the previous method

C function [Previous method]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8
short x_input[NUM] = { 0x1000, 0x2000, 0x4000,
                       0x6000, 0xf000, 0xe000,
                       0xc000, 0xa000};
short y_input[NUM] = { 0x2000, 0x4000, 0xe000,
                       0xf000, 0x6000, 0x2000,
                       0xe000, 0xf000};
short result[NUM];
void func(void)
{
    int i;
    for (i = 0; i < NUM; i++) {
        result[i] = x_input[i] - y_input[i];
    }
}

void main(void)
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
        printf("%f¥n", result[i]/32768.0);
    }
}
```

[DSP-C]

```
// -cpu=sh3dsp -dspc
#include <stdio.h>
#define NUM 8
__X __fixed x_input[NUM] = { 0.125r,   0.25r,   0.5r,
                             0.75r, -0.125r, -0.25r,
                             -0.5r, -0.75r};
__Y __fixed y_input[NUM] = {0.25r, 0.5r,   -0.25r,
                            -0.125r, 0.75r, 0.25r,
                            -0.25r, -0.125r};
__fixed result[NUM];
void func(void)
{
    int i;
    for (i = 0; i < NUM; i++) {
        result[i] = x_input[i] - y_input[i];
    }
}

void main(void)
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
        printf("%r¥n", result[i]);
    }
}
```

3.3 Saturation Qualifier

If the operation results in an overflow, saturation operation replaces the result with the largest or smallest representable value. For DSP-C, simply adding a saturation qualifier enables the saturation operation.
Use the following qualifier to specify the saturation operation:

```
_ _sat
```

You can specify the saturation qualifier only for _ _fixed or long _ _fixed data. Specifying the saturation qualifier for any other data type causes an error.
Saturation operation will be performed if an expression contains data piece for which at least one saturation qualifier (_ _sat) is specified.

Examples of use:

(1) Example of sat specification

```
_ _fixed          a;
_ _sat _ _fixed   b;
_ _fixed          c;

a = -0.75r ;
b = -0.75r ;
c = a + b  ; // c = -1.0r will result.
```

(2) Comparing DSP-C and the previous method

C function [Previous method]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8
short x_input[NUM] = {0x1000, 0x2000, 0x4000,
                      0x6000, 0xf000, 0xe000,
                      0xc000, 0xa000};
short y_input[NUM] = {0x1000, 0x2000, 0x4000,
                      0x6000, 0xf000, 0xe000,
                      0xc000, 0xa000};
short result[NUM];
void func(void)
{
    int i;
    int temp;
    for (i = 0; i < NUM; i++) {
        temp = x_input[i] + y_input[i];
        if (temp > 32767) {
            temp = 32767;
        }
            else if (temp < -32768) {
            temp = -32768;
        }
        result[i] = temp;
    }
}
void main(void)
{
    int i;
    func();
     :
```

[DSP-C]

```
// -cpu=sh3dsp -dspc
#include <stdio.h>
#define NUM 8

sat __X __fixed x_input[NUM] = { 0.125r,   0.25r,   0.5r,
                                 0.75r,   -0.125r,  -0.25r,
                                 -0.5r, -0.75r};

sat__Y __fixed y_input[NUM] = { 0.125r,   0.25r,   0.5r,
                                0.75r,   0.125r, -0.25r,
                                -0.5r, -0.75r};
__fixed result[NUM];
void func(void)
{
    int i;
    for (i = 0; i < NUM; i++) {
        result[i] = x_input[i] + y_input[i];
    }
}
void main(void)
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
        printf("%r¥n", result[i]);
    }
}
```

3.4 Circular Qualifier

Use the following qualifier to specify the modulo addressing:

```
_ _circ
```

You can specify the modulo addressing for _ _fixed type one-dimensional arrays and pointers for which the memory qualifier (_ _X/_ _Y) is specified. Specifying the modulo addressing for any other conditionscauses an error.

Examples of use:

(1) Comparing DSP-C and the previous method

C function [Previous method]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8
#define BUFFER_SIZE 4
short x_input[NUM] = {0x1000, 0x2000, 0x4000, 0x6000,,
                      0xf000, 0xe000, 0xc000, 0xa000};
short y_input[BUFFER_SIZE] = {0x2000, 0x4000,
                              0x2000, 0x1000};
short result[NUM];

void func()
{
    int i;
    for (i = 0; i < NUM; i++) {
        result[i] = x_input[i] +
        y_input[i%(BUFFER_SIZE)];
    }
}

void main()
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
            printf("%f¥n", result[i]/32768.0);
    }
}
```

[DSP-C]

```
// -cpu=sh3dsp -dspc
#include <stdio.h>
#include <machine.h>
#define NUM 8
#define BUFFER_SIZE 4
__X __fixed x_input[NUM] = { 0.125r,   0.25r,   0.5r,
                             0.75r, -0.125r, -0.25r, -0.5r,
                             -0.75r};
__circ __Y __fixed y_input[BUFFER_SIZE] = {0.25r, 0.5r,
                                           0.25r, 0.125r};
__fixed result[NUM];

void func()
{
    int i;
    set_circ_y(y_input, sizeof(y_input));
    for (i = 0; i < NUM; i++) {
            result[i] = x_input[i] + y_input[i];
    }
    clr_circ();
}

void main()
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
            printf("%r¥n", result[i]);
    }
}
```

Important Information:

(1) The modulo addressing is applicable to one-dimensional arrays and pointers that exist between the built-in functions clr_circ() and set_circ_x() or set_circ_y().
(2) Correct operation is not guaranteed if you specify the modulo addressing for multiple arrays concurrently or if you reference an array or pointer with _ _circ specified in other than between the built-in functions shown above.
(3) Correct operation is not guaranteed if you specify the modulo addressing in a negative direction.
(4) Data subject to modulo addressing must be aligned so that the higher 16 bits will be the same during liking. You cannot directly reference the contents of an array.
(5) Correct operation is not guaranteed if one of the following occurs (a warning may be output):

- optimize=0 is specified.
- The _ _circ pointer is specified for other than a local variable.
- volatile is specified for the _ _circ pointer.
- The _ _circ pointer is updated but is not referenced.
- There is a function all between the built-in functions clr_circ and set_circ_x or set_circ_y.

3.5 Type Conversion

Table 3.3 shows the rules for type conversion.

Table 3.3 Rules for Type Conversion

Conversion	Specifications
_ _fixed -> long _ _fixed _ _accum -> long _ _accum	Lower 16 bits are cleared to zero. The value remains unchanged.
long _ _fixed -> _ _fixed long _ _accum -> _ _accum	Lower 16 bits are truncated. Precision of the fractional part is degraded.
_ _fixed -> _ _accum long _ _fixed -> long _ _accum	Sign expansion is performed for higher 8 bits. The value remains unchanged.
_ _fixed -> long _ _accum	Sign expansion is performed for higher 8 bits. Lower 16 bits are cleared to zero. The value remains unchanged.
long _ _fixed -> _ _accum	Sign expansion is performed for higher 8 bits. Lower 16 bits are truncated. Precision of the fractional part is degraded.
_ _accum -> _ _fixed long _ _accum -> long _ _fixed _ _accum -> long _ _fixed	Higher 8 bits are truncated. The 9th bit must be the sign bit. The value remains unchanged if the integer part is zero.
long _ _accum -> _ _fixed	Higher 8 bits and lower 16 bits are truncated. The 9th bit must be the sign bit. The value remains unchanged if the integer part is zero. Precision of the fractional part is degraded.
_ _fixed -> signed integer type long _ _fixed -> signed integer type	The value is -1 for -1.0r and -1.0R, or 0 for other cases.
_ _accum -> signed integer type long _ _accum -> signed integer type	The fractional part is truncated. The value after conversion is an integer from -256 to 255.
_ _fixed -> unsigned integer type	For -1.0r and -1.0R, the maximum value for the type after conversion is assumed. For other cases,

Conversion	Specifications
long _ _fixed -> unsigned integer type	0 is assumed.
_ _accum -> unsigned integer type	The fractional part is truncated. For a positive value, the value after conversion is an integer from 0 to 255. For a negative value, (the value before conversion + 1 + the maximum value for the type after conversion) is assumed.
long _ _accum -> unsigned integer type	
signed integer type -> _ _fixed	The highest bit before conversion must be the highest bit after conversion. All the other bits will be zero.
signed integer type ->long _ _fixed	
signed integer type -> _ _accum	Lower 9 bits of the value must be the integer part. The fractional part must be zero.
signed integer type ->long _ _accum	
unsigned integer type -> _ _fixed	All the bits after conversion must be zero.
unsigned integer type ->long _ _fixed	
unsigned integer type -> _ _accum	Lower 9 bits of the value must be the integer part. The fractional part must be zero.
unsigned integer type ->long _ _accum	
Fixed-point -> floating-point	A value representable in the type after conversion will be the same as the original value. The value that cannot be represented is rounded to a nearest value.
Floating-point -> fixed point	The handling of the fractional part is the same as for the conversion from fixed-point to floating point. The handling of the integer part is the same as for the conversion from floating-point to integer. If the integer part is the representable range for the fixed-point, the value remains unchanged. If the integer part exceeds the range, the lowest bit of the overflow must be a sign bit. The saturation processing is not performed even if it is specified for the type after conversion.

Important Information:

(1) Conversion from (long)_ _fixed to the integer type, and vice versa
Integers that can be represented in the (long)_ _fixed type are 0 and -1.
This means that the above conversion causes missing information.

(2) Conversion from (long)_ _accum to the integer type, and vice versa
Integers in the range from -256 to 255 can be represented in the (long)_ _accum type. Integers within this range retain information after they are converted.
However, note that converting a negative value to the unsigned integer type causes an overflow.
For a series of operations that only require the integer type, conversion to the integer type may improve performance.

(3) Bit pattern copy
If you use a substitute operator to copy a bit pattern, a type conversion occurs and the expected results cannot be acquired. In this case, use the built-in functions such as long_as_lfixed and lfixed_as_long.

Website and Support <website and support,ws>

Renesas Technology Website
http://japan.renesas.com/

Inquiries
http://japan.renesas.com/inquiry
csc@renesas.com

Revision Record <revision history,rh>

Rev.	Date	Description	
		Page	Summary
1.00	Jun.1.07	—	First edition issued