

4 DSP RUN-TIME LIBRARY

Overview

The run-time library for ADSP-219x processors contains a collection of functions that provide services commonly required by DSP applications. These functions are in addition to the C/C++ run-time library functions that are described in [Chapter 3 C/C++ Run-Time Library](#). The services provided by the DSP library functions include support for interrupt handling, signal processing, and access to hardware registers. All these services are Analog Devices extensions to ANSI standard C.

For more information on the algorithms on which many of the C library's math functions are based, see the Cody and Waite text “Software Manual for the Elementary Functions” from Prentice Hall (1980).

The sections of this chapter present the following information on the compiler:

- [“DSP Run-Time Library Guide” on page 4-2](#) contains introductory information about Analog devices special header files and built-in functions that are included with this release of the cc219x compiler.
- [“DSP Run-Time Library Reference” on page 4-22](#) contains the complete reference information for each DSP run-time library function included with this release of the cc219x compiler.

DSP Run-Time Library Guide

The DSP run-time library contains functions that you can call from your source program. This section describes how to use the library and provides information on the following topics:

- [“Calling Library Functions” on page 4-2](#)
- [“Linking DSP Library Functions” on page 4-3](#)
- [“Working with Library Source Code” on page 4-3](#)
- [“Data Types” on page 4-4](#)
- [“DSP Header Files” on page 4-4](#)

Calling Library Functions

To use a DSP library function, call the function by name and give the appropriate arguments. The names and arguments for each function appear on the function’s reference page. The reference pages appear in the section, [“DSP Run-Time Library Reference” on page 4-22](#).

Like other functions you use, library functions should be declared. Declarations are supplied in header files, as described in the section, [“Working With Library Header Files” in Chapter 3 C/C++ Run-Time Library](#).

-  The function names are C function names. If you call C run-time library functions from an assembly language program, you must use the assembly version of the function name: prefix an underscore on the name. For more information on naming conventions, see the section, [“C/C++ and Assembly Language Interface” on page 2-103](#).
-  You can use the archiver, described in the *VisualDSP++ 2.0 Linker and Utilities Manual for ADSP-21xx DSPs*, to build library archive files of your own functions.

Linking DSP Library Functions

When your C code calls a DSP run-time library function, the call creates a reference that the linker resolves when linking your program. This requires the linker to be directed to link with the DSP run-time library, `libdsp.dlb`, in the `219x\lib` directory, which is a subdirectory of the VisualDSP++ installation directory. This is done automatically when using the default Linker Description File (LDF) for ADSP-219x, as this specifies that `libdsp.dlb` will be on each link line. If not using this LDF file, then either add `libdsp.dlb` to the LDF file which is being used, or alternatively use the compiler's `'-ldsp'` switch to specify that `libdsp.dlb` is to be added to the link line.

Working with Library Source Code

The source code for some functions and macros in the DSP run-time library is provided with your VisualDSP++ software. By default, the installation program copies the source code to a subdirectory of the directory where the run-time libraries are kept, named `219x\lib\src\libdsp_src`. Each function is kept in a separate file. The filename is the name of the function with the extension `.asm` or `.c`. If you do not intend to modify any of the run-time library functions, you can delete this directory and its contents to conserve disk space.

The source code is provided so you can customize specific functions for your own needs. To modify these files, you need proficiency in ADSP-219x assembly language and an understanding of the run-time environment, as explained in [“C/C++ and Assembly Language Interface” on page 2-103](#). Before you make any modifications to the source code, copy the source code to a file with a different filename and rename the function itself. Test the function before you use it in your system to verify that it is functionally correct.



Analog Devices only supports the run-time library functions as provided.

Data Types

Functions in the DSP run-time library variously support `int`, `long`, `float`, `double` and `fract16` data types as follows:

- `int` is a 16-bit integer and `long` is a 32-bit integer
- `float` and `double` are both 32-bit floating point with a range of -3.4×10^{38} to 3.4×10^{38} .
- `fract16` is a 16-bit fractional data type (1.15 format) having a range of -1.0 to +1.0. This is defined in the C language as:

```
typedef short fract16
```

DSP Header Files

The following DSP header files are supplied with this release of the cc219x compiler.

`math.h` — Math Functions

The standard math functions have been augmented by implementations for the `float` data type, and in some cases for the `fract16` data type.

[Table 4-1](#) provides a summary of the functions defined by the `math.h` header file. Descriptions of these functions are given under the name of the `double` version in the library reference section in [Chapter 3 C/C++ Run-Time Library](#).

This header also provides prototypes for a number of additional math functions `clip`, `copysign`, `max`, `min`, and an integer function `countones`. These are described in the library reference section in this chapter.

Table 4-1. Math Library

Description	Prototype
Arc Cosine	double acos (double x); float acosf (float x); fract16 acos_fr16 (fract16 x);
Arc Sine	double asin (double x); float asinf (float x); fract16 asin_fr16 (fract16 x);
Arc Tangent	double atan (float x); float atanf (float x); fract16 atan_fr16 (fract16 x);
Arc Tangent of Quotient	double atan2 (double y, double x); float atan2f (float y, float x); fract16 atan2_fr16 (fract16 y, fract16 x);
Ceiling	double ceil (double x); float ceilf (float x);
Cosine	double cos (double x); float cosf (float x); fract16 cos_fr16 (fract16 x);
Hyperbolic Cosine	double cosh (double x); float coshf (float x);

DSP Run-Time Library Guide

Table 4-1. Math Library (Cont'd)

Description	Prototype
Cotangent	double cot (double x); float cotf (float x);
Exponential	double exp (double x); float exp (float x);
Floor	double floor (double x); real floorf (float x);
Floating Point Remainder	double fmod (double x, double y); float fmodf (float x, float y);
Get Mantissa and Exponent	double frexp (double x, int *n); float frexp (float x, int *n);
Natural Logarithm	double log (double x); float logf (float x);
Logarithm Base 10	double log10 (double x); float log10f (float x);
Get Fraction and Integer	double modf (double x, double *i); float modff (float x, float *i);
Power	double pow (double x, double y); float powf (float x, float y);
Sine	double sin (double x); float sinf (float x); fract16 sin_fr16 (fract16 x);

Table 4-1. Math Library (Cont'd)

Description	Prototype
Hyperbolic Sine	double sinh (double x); float sinhf (float x);
Square Root	double sqrt (double x); float sqrtf (float x); fract16 sqrt_fr16 (fract16 x);
Reciprocal of Square Root	double rsqrt (double x); float rsqrtf (float x);
Tangent	double tan (double x); float tanf (float x); fract16 tan_fr16 (fract16 x);
Hyperbolic Tangent	double tanh (double x); float tanhf (float x);

complex.h — Basic Complex Arithmetic Functions

The `complex.h` header file contains utility functions that provide the type definitions and basic arithmetic operations on `complex_float`, `complex_double` and `complex_fract16` variables.

The complex functions defined in the `complex.h` header file are listed in [Table 4-2 on page 4-8](#).

DSP Run-Time Library Guide

The following structures are used to represent complex numbers in rectangular coordinates:

```
typedef struct
{
    float re;
    float im;
} complex_float;

typedef struct
{
    double re;
    double im;
} complex_double;

typedef struct
{
    fract16 re;
    fract16 im;
} complex_fract_16
```

Details of the basic complex arithmetic functions are included in the library reference section of this chapter.

Table 4-2. Complex Functions

Description	Prototype
Complex Absolute Value	double cabs (complex_double a); float cabsf (complex_float a); fract16 cabs_fr16 (complex_fract16 a);
Complex Addition	complex_double cadd (complex_double a, complex_double b); complex_float caddf (complex_float a, complex_float b); complex_fract16 cadd_fr16 (complex_fract16 a, complex_fract16 b);
Complex Subtraction	complex_double csub (complex_double a, complex_double b); complex_float csubf (complex_float a, complex_float b); complex_fract16 csub_fr16 (complex_fract16 a, complex_fract16 b);

Table 4-2. Complex Functions (Cont'd)

Description	Prototype
Complex Multiply	<code>complex_double cmlt (complex_double a, complex_double b);</code> <code>complex_float cmltf (complex_float a, complex_float b);</code> <code>complex_fract16 cmlt_fr16 (complex_fract16 a, complex_fract16 b);</code>
Complex Division	<code>complex_double cdiv (complex_double a, complex_double b);</code> <code>complex_float cdivf (complex_float a, complex_float b);</code> <code>complex_fract16 cdiv_fr16 (complex_fract16 a, complex_fract16 b);</code>
Get Phase of a Complex Number	<code>double arg (complex_double a);</code> <code>float argf (complex_float a);</code> <code>fract16 arg_fr16 (complex_fr16 a);</code>
Complex Conjugate	<code>complex_double conj (complex_double a);</code> <code>complex_float conjf (complex_float a);</code> <code>complex_fract16 conj_fr1 (complex_fract16 a);</code>
Complex Polar Coordinates	<code>complex_double polar (double mag, double phase);</code> <code>complex_float polarf (float mag, float phase);</code> <code>complex_fract16 polar_fr16 (fract16 mag, fract16 phase);</code>
Complex Exponential	<code>complex_double cexp (double a);</code> <code>complex_float cexpf (float a);</code>
Normalization	<code>complex_double norm (complex_double a);</code> <code>complex_float normf (complex_float a);</code>

vector.h — Vector Functions

The `vector.h` header file contains functions for operating on real and complex vectors, both vector-scalar and vector-vector operations. See [“complex.h — Basic Complex Arithmetic Functions” on page 4-7](#) for definition of the complex types.

The vector functions in the `vector.h` header file are listed in [Table 4-3](#). In the prototypes in the table, `a[]` and `b[]` are input vectors, `b` is an input scalar, `c[]` is an output vector and `n` is the number of elements.

The functions described by this header assume that input array arguments are constant, i.e., their contents will not change during the course of the routine. In particular, this means the input arguments do not overlap with any output argument.

Table 4-3. Vector Functions

Description	Prototype
Real Vector + Scalar Addition	<code>void vecsadd (const double a [], double b, double c [], int n);</code> <code>void vecsaddf (const float a [], float b, float c [], int n);</code> <code>void vecsadd_fr16 (const fract16 a [], fract16 b, fract16 c [], int n);</code>
Real Vector - Scalar Subtraction	<code>void vecssub (const double a [], double b, double c [], int n);</code> <code>void vecssubf (const float a [], float b, float c [], int n);</code> <code>void vecssub_fr16 (const fract16 a [], fract16 b, fract16 c [], int n);</code>
Real Vector * Scalar Multiplication	<code>void vecsmult (const double a [], double b, double c [], int n);</code> <code>void vecsmultf (const float a [], float b, float c [], int n);</code> <code>void vecsmult_fr16 (const fract16 a [], fract16 b, fract16 c [], int n);</code>
Real Vector + Vector Addition	<code>void vecvadd (const double a [], const double b [], double c [], int n);</code> <code>void vecvaddf (const float a [], const float b [], float c [], int n);</code> <code>void vecvadd_fr16 (const fract16 a [], const fract16 b [], fract16 c [], int n);</code>

Table 4-3. Vector Functions (Cont'd)

Description	Prototype
Real Vector - Vector Subtraction	void vecvsub (const double a [], const double b [], double c [], int n); void vecvsubf (const float a [], const float b [], float c [], int n); void vecvsub_fr16 (const fract16 a [], const fract16 b [], fract16 c [], int n);
Real Vector * Vector Multiplication	void vecvmlt (const double a [], const double b [], double c [], int n); void vecvmltf (const float a [], const float b [], float c [], int n); void vecvmlt_fr16 (const fract16 a [], const fract16 b [], fract16 c [], int n);
Maximum Value of Vector Elements	double vecmax (const double a [], int n); float vecmaxf (const float a [], int n); fract16 vecmax_fr16 (const fract 16 a [], int n);
Minimum Value of Vector Elements	double vecmin (const double a [], int n); float vecminf (const float a [], int n); fract16 vecmin_fr16 (const fract16 a [], int n);
Index of Maximum Value of Vector Elements	int vecmaxloc (const double a [], int n); int vecmaxlocf (const float a [], int n); fract16 vecmaxloc_fr16 (const fract 16 a [], int n);
Index of Minimum Value of Vector Elements	int vecminloc (const double a [], int n); int vecminlocf (const float a [], int n); fract16 vecminloc_fr16 (const fract 16 a [], int n);
Complex Vector + Scalar Addition	void cvecsadd (const complex_double a [], complex_double b, complex_double c [], int n); void cvecsaddf (const complex_float a [], complex_float b, complex_float c [], int n); void cvecsadd_fr16 (const complex_fract16 a [], complex_fract16 b, complex_fract16 c [], int n);

DSP Run-Time Library Guide

Table 4-3. Vector Functions (Cont'd)

Description	Prototype
Complex Vector - Scalar Subtraction	<pre>void cvecssub (const complex_double a [], complex_double b, complex_double c [], int n); void cvecssubf (const complex_float a [], complex_float b, complex_float c [], int n); void cvecssub_fr16 (const complex_fract16 a [], complex_fract16 b, complex_fract16 c [], int n);</pre>
Complex Vector * Scalar Multiplication	<pre>void cvecsmult (const complex_double a [], complex_double b, complex_double c [], int n); void cvecsmultf (const complex_float a [], complex_float b, complex_float c [], int n); void cvecsmult_fr16 (const complex_fract16 a [], complex_fract16 b, complex_fract16 c [], int n);</pre>
Complex Vector + Vector Addition	<pre>void cvecvadd (const complex_double a [], const complex_double b [], complex_double c [], int n); void cvecvaddf (const complex_float a [], const complex_float b [], complex_float c [], int n); void cvecvadd_fr16 (const complex_fract16 a [], const complex_fract16 b [], complex_fract16 c [], int n);</pre>
Complex Vector - Vector Subtraction	<pre>void cvecvsub (const complex_double a [], const complex_double b [], complex_double c [], int n); void cvecvsubf (const complex_float a [], const complex_float b [], complex_float c [], int n); void cvecvsub_fr16 (const complex_fract16 a [], const complex_fract16 b [], complex_fract16 c [], int n);</pre>

Table 4-3. Vector Functions (Cont'd)

Description	Prototype
Complex Vector * Vector Multiplication	<pre>void cvecvmlt (const complex_double a [], const complex_double b [], complex_double c [], int n); void cvecvmltf (const complex_float a [], const complex_float b [], complex_float c [], int n); void cvecvmlt_fr16 (const complex_fract16 a [], const complex_fract16 b [], complex_fract16 c [], int n);</pre>
Real Vector Dot Product	<pre>double vecdot (const double a [], const double b [], int n); float vecdotf (const float a [], const float b [], int n); fract16 vecdot_fr16 (const fract16 a [], const fract16 b [], int n);</pre>
Complex Vector Dot Product	<pre>complex_double cvecdot (const complex_double a [], const complex_double b [], int n); complex_float cvecdotf (const complex_float a [], const complex_float b [], int n); complex_fract16 cvecdot_fr16 (const complex_fract16 a [], const complex_fract16 b [], complex_fract16 c [], int n);</pre>

matrix.h — Matrix Functions

The `matrix.h` header file contains matrix functions for operating on real and complex matrices, both matrix-scalar and matrix-matrix operations. See [“complex.h — Basic Complex Arithmetic Functions” on page 4-7](#) for definition of the complex types.

The matrix functions defined in the `matrix.h` header file are listed in [Table 4-4](#). In most of the function prototypes `*a` is a pointer to input matrix `a [ ] [ ]`, `*b` is a pointer to input matrix `b [ ] [ ]`, `b` is an input scalar, `n` is the number of rows, `m` is the number of columns, and `*c` is a pointer to output matrix `c [ ] [ ]`. In the `matrix*matrix` functions, `n` and `k` are the dimensions of matrix `a`, and `k` and `m` are the dimensions of matrix `b`.

DSP Run-Time Library Guide

The functions described by this header assume that input array arguments are constant, i.e., their contents do not change during the course of the routine. In particular, this means the input arguments do not overlap with any output argument.

Table 4-4. Matrix Functions

Description	Prototype
Real Matrix + Scalar Addition	<code>void matsadd (const double *a, const double b, int n, int m, double *c);</code> <code>void matsaddf (const float *a, const float b, int n, int m, float *c);</code> <code>void matsadd_fr16 (const fract16 *a, const fract16 b, int n, int m, fract16 *c);</code>
Real Matrix - Scalar Subtraction	<code>void matssub (const double *a, const double b, int n, int m, double *c);</code> <code>void matssubf (const float *a, const float b, int n, int m, float *c);</code> <code>void matssub_fr16 (const fract16 *a, const fract16 b, int n, int m, fract16 *c);</code>
Real Matrix * Scalar Multiplication	<code>void matsmlt (const double *a, double b, int n, int m, double *c);</code> <code>void matsmltf (const float *a, float b, int n, int m, float *c);</code> <code>void matsmlt_fr16 (const fract16 *a, fract16 b, int n, int m, fract16 *c);</code>
Real Matrix + Matrix Addition	<code>void matmadd (const double *a, double *b, int n, int m, double *c);</code> <code>void matmaddf (const float *a, float *b, int n, int m, float *c);</code> <code>void matmadd_fr16 (const fract16 *a, fract16 *b, int n, int m, fract16 *c);</code>
Real Matrix - Matrix Subtraction	<code>void matmsub (const double *a, double *b, int n, int m, double *c);</code> <code>void matmsubf (const float *a, float *b, int n, int m, float *c);</code> <code>void matmsub_fr16 (const fract16 *a, fract16 *b, int n, int m, fract16 *c);</code>
Real Matrix * Matrix Multiplication	<code>void matmmlt (const double *a, int n, int k, const double *b, int m, double *c);</code> <code>void matmmltf (const float *a, int n, int k, const float *b, int m, float *c);</code> <code>void matmmlt_fr16 (const fract16 *a, int n, int k, const fract16 *b, int m, fract16 *c);</code>

Table 4-4. Matrix Functions (Cont'd)

Description	Prototype
Complex Matrix + Scalar Addition	void cmatsadd (const complex_double *a, complex_double b, int n, int m, complex_double *c); void cmatsaddf (const complex_float *a, complex_float b, int n, int m, complex_float *c); void cmatsadd_fr16 (const complex_fract16 *a, complex_fract16 b, int n, int m, complex_fract16 *c);
Complex Matrix - Scalar Subtraction	void cmatssub (const complex_double *a, complex_double b, int n, int m, complex_double *c); void cmatssubf (const complex_float *a, complex_float b, int n, int m, complex_float *c); void cmatssub_fr16 (const complex_fract16 *a, complex_fract16 b, int n, int m, complex_fract16 *c);
Complex Matrix * Scalar Multiplication	void cmatsmlt (const complex_double *a, complex_double b, int n, int m, complex_double *c); void cmatsmltf (const complex_float *a, complex_float b, int n, int m, complex_float *c); void cmatsmlt_fr16 (const complex_fract16 *a, complex_fract16 b, int n, int m, complex_fract16 *c);
Complex Matrix + Matrix Addition	void cmatmadd (const complex_double *a, const complex_double *b, int n, int m, complex_double *c); void cmatmaddf (const complex_float *a, const complex_float *b, int n, int m, complex_float *c); void cmatmadd_fr16 (const complex_fract16 *a, const complex_fract16 *b, int n, int m, complex_fract16 *c);
Complex Matrix – Matrix Subtraction	void cmatmsub (const complex_double *a, const complex_double *b, int n, int m, complex_double *c); void cmatmsubf (const complex_float *a, const complex_float *b, int n, int m, complex_float *c); void cmatmsub_fr16 (const complex_fract16 *a, const complex_fract16 *b, int n, int m, complex_fract16 *c);

Table 4-4. Matrix Functions (Cont'd)

Description	Prototype
Complex Matrix * Matrix Multiplication	void cmatmmlt (const complex_double *a, int n, int k, const complex_double *b, int m, complex_double *c); void cmatmmltf (const complex_float *a, int n, int k, const complex_float *b, int m, complex_float *c); void cmatmmlt_fr16 (const complex_fract16 *a, int n, int k, const complex_fract16 *b, int m, complex_fract16 *c);
Transpose	void transpm (const double *a, int n, int m, double *c); void transpmf (const float *a, int n, int m, float *c); void transpm_fr16 (const fract16 *a, int n, int m, fract16 *c);

filter.h — DSP Filters and Transformations

The `filter.h` header file contains various filters used in digital signal processing, including `FIR`, `IIR`, and so forth. It also contains functions that perform key transformations used in DSPs, including `FFT` and `convolve`. A-law and μ -law companders are also provided. See [“complex.h — Basic Complex Arithmetic Functions” on page 4-7](#) for definition of the complex types. The functions defined in the `filter.h` header file are listed in [Table 4-5 on page 4-17](#) and [Table 4-6 on page 4-17](#) and are described in detail in the library reference section of this chapter.

For efficiency, the twiddle table is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

Various different versions of the FFT are provided, including `cfft`, `rfft`, `ifft`, `cfft2d`, `rfft2d`, `ifft2d`. The number of points is provided as an argument; when appropriate, the library uses radix 2 or radix 4 implementations.

Several FFT's of different sizes can all be accommodated with the same twiddle factor table. To do this, allocate the table at the maximum size. Each FFT has an additional parameter, the “stride” of the twiddle table. To use the whole table, specify a stride of 1. If your FFT uses only half the points of the largest, the stride should be 2 (this takes only every other element).

Table 4-5. Filter Library

Description	Prototype
Finite Impulse Response Filter	<code>void fir_fr16 (const fract16 x[], fract16 y[], int n, fir_state_fr16 *s);</code>
Infinite Impulse Response Filter	<code>void iir_fr16 (const fract16 x[], fract16 y[], int n, iir_state_fr16 *s);</code>
Fir Decimation Filter	<code>void fir_decima_fr16 (const fract16 x[], fract16 y[], int n, fir_state_fr16 *s);</code>
Fir Interpolation Filter	<code>void fir_interp_fr16 (const fract16 x[], fract16 y[], int n, fir_state_fr16 *s);</code>
Complex Finite Impulse Response Filter	<code>void cfir_fr16 (const complex_fract16 x[], complex_fract16 y[], int n, cfir_state_fr16 *s);</code>

Table 4-6. Transformation Functions

Description	Prototype
Fast Fourier Transform	
Generate FFT Twiddle Factor	<code>void twidfft_fr16 (complex_fract16 w[], int n);</code>
N Point Radix 2 Complex Input FFT	<code>void cfft_fr16 (const complex_fract16 *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int wst, int n, int block_exponent, int scale_method);</code>

Table 4-6. Transformation Functions (Cont'd)

Description	Prototype
N Point Radix 2 Real Input FFT	<code>void rfft_fr16 (const fract16 *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int mst, int n, int block_exponent, int scale_method);</code>
N Point Radix 2 Inverse FFT	<code>void ifft_fr16 (const complex_fract *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int wst, int n, int block_exponent, int scale_method);</code>
N Point Radix 4 Complex Input FFT	<code>void cfft4d_fr16 (const complex_fract16 *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int wst, int n, int block_exponent, int scale_method);</code>
N Point Radix 4 Real Input FFT	<code>void rfft4d_fr16 (const fract16 *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int mst, int n, int block_exponent, int scale_method);</code>
N Point Radix 4 Inverse Input FFT	<code>void ifft4d_fr16 (const complex_fract *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int wst, int n, int block_exponent, int scale_method);</code>
Nxn Point 2-D Complex Input FFT	<code>void cfft2d_fr16 (const complex_fract16 *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int wst, int n, int block_exponent, int scale_method);</code>
Nxn Point 2-D Real Input FFT	<code>void rfft2d_fr16 (const fract16 *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int mst, int n, int block_exponent, int scale_method);</code>
Nxn Point 2-D Inverse FFT	<code>void ifft2d_fr16 (const complex_fract *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int wst, int n, int block_exponent, int scale_method);</code>
Convolutions	
Convolution	<code>void convolve_fr16 (const fract16 cin1[], int clen1, const fract16 cin2[], int clen2, fract16 cout[]);</code>
2-D Convolution	<code>void conv2d_fr16 (const fract16 *cin1, int crow1, int ccol1, const fract16 *cin2, int crow2, int ccol2, fract16 *cout);</code>

Table 4-6. Transformation Functions (Cont'd)

Description	Prototype
2-D Convolution 3x3 Matrix	<code>void conv2d3x3_fr16 (const fract16 *cin, int crow1, int ccol1, const fract16 cin2 [3] [3], fract16 *cout);</code>
Compression/Expansion	
a-law Compression	<code>void a_compress (const short in[], short out[], int n);</code>
a-law Expansion	<code>void a_expand (const char in[], short out[], int n);</code>
M-law Compression	<code>void mu_compress (const short in[], short out[], int n);</code>
M-law Expansion	<code>void mu_expand (const char in[], short out[], int n);</code>

stats.h — Statistical Functions

The statistical functions defined in the `stats.h` header file are listed in [Table 4-7](#) and are described in detail in the library reference section of this chapter.

Table 4-7. Stats Functions

Description	Prototype
Autocoherence	<code>void autocohf (const float a[], int n, int m, float c[]);</code> <code>void autocoh_fr16 (const fract16 a[], int n, int m, fract16 c[]);</code>
Autocorrelation	<code>void autocorrf (const float a[], int n, int m, float c[]);</code> <code>void autocorr_fr16 (const fract16 a[], int n, int m, fract16 c[]);</code>
Cross-coherence	<code>void crosscohf (const float a[], const float b[], int n, int m, float c[]);</code> <code>void crosscoh_fr16 (const fract16 a[], const fract16 b[], int n, int m, fract16 c[]);</code>

Table 4-7. Stats Functions (Cont'd)

Description	Prototype
Cross-correlation	<code>void crosscorr_f (const float a[], const float b[], int n, int m, float c[]);</code> <code>void crosscorr_fr16 (const fract16 a[], const fract16 b[], int n, int m, fract16 c[]);</code>
Histogram	<code>void histogram_f (const float a[], int c[], float max, float min, int n, int m);</code> <code>void histogram_fr16 (const fract16 a[], int c[], fract16 max, fract16 min, int n, int m);</code>
Mean	<code>float mean_f (const float a[], int n);</code> <code>fract16 mean_fr16 (const fract16 a[], int n);</code>
Root Mean Square	<code>float rms_f (const float a[], int n);</code> <code>fract16 rms_fr16 (const fract16 a[], int n);</code>
Variance	<code>float var_f (const float a[], int n);</code> <code>fract16 var_fr16 (const fract16 a[], int n);</code>
Count Zero Crossing	<code>float zero_cross_f (const float a[], int n);</code> <code>fract16 zero_cross_fr16 (const fract16 a[], int n);</code>

window.h — Window Generators

Contains various functions to generate windows based on various methodologies. The functions defined in the `window.h` header file are listed in [Table 4-8](#) and are described in detail in the library reference section of this chapter.

For all window functions, a stride parameter *a* can be used to space the window values. The window length parameter *n* equates to the number of elements in the window. Therefore, for a stride *a* of 2 and a length *n* of 10, an array of length 20 is required, where every second entry is untouched.

Table 4-8. Window Generator Functions

Description	Prototype
Generate Bartlett Window	<code>void gen_bartlett_fr16 (fract16 w[], int a, int n);</code>
Generate Blackman Window	<code>void gen_blackman_fr16 (fract16 w[], int a, int n);</code>
Generate Gaussian Window	<code>void gen_gaussian_fr16 (fract16 w[], float alpha, int a, int n);</code>
Generate Hamming Window	<code>void gen_hamming_fr16 (fract16 w[], int a, int n);</code>
Generate Hann Window	<code>void gen_hann_fr16 (fract16 w[], int a, int n);</code>
Generate Hanning Window	<code>void gen_hanning_fr16 (fract16 w[], int a, int n);</code>
Generate Harris Window	<code>void gen_harris_fr16 (fract16 w[], int a, int n);</code>
Generate Kaiser Window	<code>void gen_kaiser_fr16 (fract16 w[], float beta, int a, int n);</code>
Generate Rectangular Window	<code>void gen_rectangular_fr16 (fract16 w[], int a, int n);</code>
Generate Triangle Window	<code>void gen_triangle_fr16 (fract16 w[], int a, int _n);</code>
Generate Vonhann Window	<code>void gen_vonhann_fr16 (fract16 _w[], int a, int n);</code>

DSP Run-Time Library Reference

The DSP run-time library is a collection of functions that you can call from your source programs.

 The information that follows applies to all of the functions in the library.

Notation Conventions. An interval of numbers is indicated by the minimum and maximum, separated by a comma, and enclosed in two square brackets, two parentheses, or one of each. A square bracket indicates that the endpoint is included in the set of numbers; a parenthesis indicates that the endpoint is not included.

Reference Format. Each function in the library has a reference page. These pages have the following format:

Name and Purpose of the function

Synopsis—indicates which header file should be `#include'd` for this function and provides C prototype (as it is found in the header) describing the interface to the function. When the functionality is provided for several data types, such as `float`, `double`, and `fract16`, several prototypes are given.

 For some functions, the interface is presented using the "K&R" style for ease of documentation. A true C prototype is provided in the header file.

Description—Description of function and parameters

Algorithm—High level mathematical representation of the function

Domain—Range of values supported by function

Notes—Miscellaneous information of note

a_compress

A-law compression

Synopsis

```
#include <filter.h>
void a_compress(in, out, n)
const short in[];    /* Input array */
short out[];         /* Output array */
int n;               /* number of elements to be compressed */
```

Description

This function performs A-law compression (ITU rec. G.711) on the elements of the input vector pointed to by `in` and outputs compressed data in the vector pointed to by `out`.

Algorithm

$C(k)$ =a-law compression of $A(k)$ for $k=0$ to n

Domain

content of input array: -8192 to 8191

a_expand

A-law expansion

Synopsis

```
#include <filter.h>
void a_expand(in, out, n)
short char in[];      /* Input array */
short out[];          /* Output array */
int n;                /* number of elements to be expanded */
```

Description

This function performs a-law expansion (ITU rec. G.711) on the elements of the input vector pointed to by `in` and outputs expanded data in vector pointed to by `out`.

Algorithm

$C(k)$ =a-law expansion of $A(k)$ for $k=0$ to $n-1$

Domain

content of input array: 0 to 255

arg

get phase of a complex number

Synopsis

```
#include <complex.h>
float argf (complex_float a)
double arg (complex_double a)
fract16 arg_fr16 (complex_fract16 a)
```

Description

This function computes the phase of complex input *a* and returns the result.

Algorithm

$$c = \operatorname{atan}\left(\frac{\operatorname{Im}(a)}{\operatorname{Re}(a)}\right)$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for `argf( )`, `arg( )`

-1.0 to +1.0 for `arg_fr16( )`

Note

$\operatorname{Im}(a) / \operatorname{Re}(a) \leq 1$ for `arg_fr16( )`

autocoh

autocoherence

Synopsis

```
#include <stats.h>
void autocohf(a,n,m,c)
const float a[];          /* Input vector a */
int n;                    /* Input samples */
int m;                    /* Lag count */
float c[];                 /* Output vector c */
void autocoh_fr16 (a,n,m,c)
const fract16 a[];        /* Input vector a */
int n;                    /* Input samples */
int m;                    /* Lag count */
fract16 c[];              /* Output vector c */
```

Description

This function computes the autocohereence of the input elements contained within input vector a, and stores the result to output vector c.

Algorithm

$$c_k = \frac{1}{n} * \left(\sum_{j=0}^{n-k-1} (a_j - \bar{a}) * (a_{j+k} - \bar{a}) \right)$$

where k={0,1,...,m-1} and a is the mean value of input vector a.

Domain

-3.4 x 10³⁸ to +3.4 x 10³⁸ for autocohf()

-1.0 to 1.0 for autocoh_fr16()

autocorr

autocorrelation

Synopsis

```
#include <stats.h>
void autocorrf(a,n,m,c)
const float a[];      /* Input vector a */
int n;                /* Number of input samples */
int m;                /* Lag count */
float c[];            /* Output vector c */
void autocorr_fr16 (a,n,m,c)
const fract16 a[];    /* Input vector a */
int n;                /* Number of input samples*/
int m;                /* Lag count */
fract16 c[];          /* Output vector c */
```

Description

This function computes the autocorrelation of the input elements contained within input vector *a*, and stores the result to output vector *c*.

Algorithm

$$c_k = \frac{1}{n} * \left(\sum_{j=0}^{n-k-1} a_j * a_{j+k} \right)$$

where $k=\{0,1,...,m-1\}$

Domain

-3.4 x 10³⁸ to +3.4 x 10³⁸ for autocorrf()

-1.0 to + 1.0 for autocorr_fr16()

DSP Run-Time Library Reference

cabs

complex absolute value

Synopsis

```
#include <complex.h>
float cabsf (complex_float a)
double cabs (complex_double a)
fract16 cabs_fr16 (complex_fract16 a)
```

Description

This function computes the complex absolute value of a complex input and returns the result.

Algorithm

$$c = \sqrt{\text{Re}^2(a) + \text{Im}^2(a)}$$

Domain

$\text{Re}^2(a) + \text{Im}^2(a) \leq 3.4 \times 10^{38}$ for `cabsf( )`, `cabs( )`

$\text{Re}^2(a) + \text{Im}^2(a) \leq 1.0$ for `cabs_fr16`

Note

The minimum input value for both real and imaginary parts can be less than 1/256 for `cabs_fr16` but the result may have bit error of 2-3 bits.

cadd

complex addition

Synopsis

```
#include <complex.h>
complex_float caddf (complex_float a, complex_float b)
complex_double cadd (complex_double a, complex_double b)
complex_fract16 cadd_fr16 (complex_fract16 a, complex_fract16 b)
```

Description

This function computes the complex addition of two complex inputs: *a*, and *b*, and returns the result.

Algorithm

$$\text{Re}(c) = \text{Re}(a) + \text{Re}(b)$$

$$\text{Im}(c) = \text{Im}(a) + \text{Im}(b)$$
Domain

-3.4 x 10³⁸ to +3.4 x 10³⁸ for caddf(), cadd()

-1.0 to +1.0 for cadd_fr16()

cdiv

complex division

Synopsis

```
#include <complex.h>
complex_float cdivf (complex_float a, complex_float b)
complex_double cdiv (complex_double a, complex_double b)
complex_fract16 cdiv_fr16 (complex_fract16 a, complex_fract16 b)
```

Description

This function computes the complex division of two complex inputs: *a* and *b*, and returns the result.

Algorithm

$$\text{Re}(c) = \frac{\text{Re}(a) * \text{Re}(b) + \text{Im}(a) * \text{Im}(b)}{\text{Re}^2(b) + \text{Im}^2(b)}$$
$$\text{Im}(c) = \frac{\text{Re}(b) * \text{Im}(a) - \text{Im}(b) * \text{Re}(a)}{\text{Re}^2(b) + \text{Im}^2(b)}$$

Domain

-3.4 x 10 ³⁸ to +3.4 x 10 ³⁸	for cdivf(), cdiv()
-1.0 to 1.0	for cdiv_fr16()

cexp

complex exponential

Synopsis

```
#include <complex.h>
complex_float cexpf (float a)
complex_double cexp (double a)
```

Description

This function computes and returns the complex exponential of real input *a* and stores the result in complex output *c*.

Algorithm

$$\text{Re}(c) = \cos(a)$$

$$\text{Im}(c) = \sin(a)$$

Domain

a = [-9099 ... 9099] for cexpf(), cexp()

cfft

N point complex input FFT

Synopsis

```
#include <filter.h>

void cfft_fr16(in[], t[], out[], w[], wst, n, block_exponent,
scale_mc)
const complex_fract16 in[] /* input sequence */
complex_fract16 t[];      /* temporary working buffer */
complex_fract16 out[];    /* output sequence */
const complex_fract16 w[]; /* twiddle sequence */
int wst;                  /* twiddle factor stride */
int n;                    /* number of FFT points */
int block_exponent;      /* block exponent of output data */
int scale_method;        /* scaling method desired 0-none,
                           1-static, 2-dynamic */
```

Description

This function transforms the time domain complex input signal sequence to the frequency domain by using the radix-2 'Fast Fourier Transform'. If input data can be overwritten, the optimum memory usage can be achieved by setting the output pointer to the input array.

For efficiency, the “twiddle table” is calculated once, during initialization, and then provided to the `cfft` routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

If the twiddle table has been allocated at a larger size than needed for a particular call of `cfft`, then the stride parameter will need to be set appropriately; otherwise, it should be one. [For more information, see “twidfft” on page 4-90.](#)

Algorithm

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk}$$

When the sequence length, n , equals power of four, the `cfftrad4` algorithm is available as well.

Domain

Input sequence length n must be a power of two and at least 16.

cffttrad4

N point complex input FFT

Synopsis

```
#include <filter.h>

void cffttrad4_fr16 (in[], t[], out[], w[], wst, n,
                    block_exponent, scale_mc)
const complex_fract16 in[]; /* input sequence */
complex_fract16 t[]; /* temporary working buffer */
complex_fract16 out[]; /* output sequence */
const complex_fract16 w[]; /* twiddle sequence */
int wst; /* twiddle factor stride */
int n; /* number of FFT points */
int block_exponent; /* block exponent of output data */
int scale_method; /* scaling method desired
                  0-none, 1-static, 2-dynamic */
```

Description

This function transforms the time domain complex input signal sequence to the frequency domain by using the radix-4 'Fast Fourier Transform'. The CFFTTRAD4 function will 'decimate in frequency' by the radix-4 FFT algorithm. If input data can be overwritten, the optimum memory usage can be achieved by setting the output pointer to the input array.

For efficiency, the “twiddle table” is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

If the twiddle table has been allocated at a larger size than needed for a particular call of `cffttrad4`, then the stride parameter will need to be set appropriately; otherwise, it should be one. [For more information, see “twidfft” on page 4-90.](#)

The temporary working buffer `t` parameter needs to be the same length as the number of FFT points, ie parameter `n`.

Algorithm

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk}$$

When the sequence length, `n`, is not a power of four, the radix2 method must be used. See CFFT.

Domain

Input sequence length `n` must be a power of four, and at least 16.

DSP Run-Time Library Reference

cfft2d

NxN point 2-D complex input FFT

Synopsis

```
#include <filter.h>

void cfft2d_fr16(*in, *t, *out, w[], wst, n, block_exponent,
scale_mc)
const complex_fract16 *in; /* pointer to input matrix
                             a[n][n]*/
complex_fract16*t;         /* pointer to working buffer
                             t[n][n] */
complex_fract16 *out;      /* pointer to output matrix
                             c[n][n] */
const complex_fract16 w[]; /* twiddle sequence */
int wst;                   /* twiddle factor stride */
int n;                     /* number of FFT points */
int block_exponent;       /* block exponent of output data */
int scale_method;         /* scaling method desired 0-none,
                             1-static, 2-dynamic */
```

Description

This function computes the two dimensional Fast Fourier Transform of the complex input matrix `a[n][n]`, and stores the result to the complex output matrix `c[n][n]`.

If the input data can be overwritten, optimum memory usage is achieved by setting the output pointer to the input array.

For efficiency, the “twiddle table” is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

If the twiddle table has been allocated at a larger size than needed for a particular call of `cfft2d`, then the stride parameter will need to be set appropriately; otherwise, it should be one. [For more information, see “twidfft” on page 4-90.](#)

Algorithm

$$c(i, j) = \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} a(k, l) * e^{-2\pi j(i*k + j*l)/n}$$

where $i=\{0,1,\dots,n-1\}$, $j=\{0,1,2,\dots,n-1\}$

Domain

Input sequence length n must be a power of two and at least 16.

cfirc

finite impulse response filter

Synopsis

```
#include <filter.h>

void fir_fr16(x,y,n,s)
    const complex_fract16 x[]; /* Input sample vector x */
    complex_fract16 y[];      /* Output sample vector y */
    int n;                    /* Number of input samples */
    fir_state *s              /* Pointer to filter state
                               structure */
```

The FIR filter function uses the following structure to maintain the state of the filter:

```
typedef struct
{
    complex_fract16 *h, /* filter coefficients */
    complex_fract16 *d, /* start of delay line */
    complex_fract16 *p, /* read/write pointer */
    int k,              /* no. of coefficients */
    int l               /* interpolation/decimation index */
} fir_state_fr16;

void cfir_init_fr16(cfir_state_fr16 *s, complex_fract16 h[],
complex_fract16 d[], int k, int l);
```

Description

This function computes a C FIR (Complex Finite Impulse Response Filter) using the coefficients stored in the vector `s->h->re` and `s->h->im` respectively for the real and imaginary parts, applied to the elements of complex input vector `x`. The results are stored in complex vector `y`.

For efficiency, this filter uses a separate variable of type `cfir_state_fr16` to maintain the filter state. You must declare space for this variable, and initialize it prior to using the filter. An initialization function, `cfir_init_fr16` is provided.

Algorithm

$$y(k) = \sum_{i=0}^{p-1} h(i) * x(k-i) \text{ for } k = 0, 1, \dots, n$$

Domain

-1.0 to +1.0

DSP Run-Time Library Reference

clip

clip

Synopsis

```
#include <math.h>
int clip (int parm1, int parm2)
float fclipf (float parm1, float parm2)
double fclip (double parm1, double parm2)
fract16 clip_fr16 (fract16 parm1, fract16 parm2)
```

Description

Clip a value if it is too large.

Algorithm

```
If (|parm1| < |parm2|)
return(parm1)
else
return(|parm2| * signof(parm1))
```

Domain

Full Range for various input parameter types.

cmlt

complex multiply

Synopsis

```
#include <complex.h>
complex_float cmltf (complex_float a, complex_float b)
complex_double cmlt (complex_double a, complex_double b)
complex_fract16 cmlt_fr16 (complex_fract16 a, complex_fract16 b)
```

Description

This function computes the complex multiply of two complex inputs: *a*, *b*, and returns the result.

Algorithm

$$\begin{aligned}\text{Re}(c) &= \text{Re}(a) * \text{Re}(b) - \text{Im}(a) * \text{Im}(b) \\ \text{Im}(c) &= \text{Re}(a) * \text{Im}(b) + \text{Im}(a) * \text{Re}(b)\end{aligned}$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for `cmltf( )`, `cmlt( )`

-1.0 to 1.0 for `cmlt_fr16( )`

conj

complex conjugate

Synopsis

```
#include <complex.h>
complex_float conjf (complex_float a, complex_float b)
complex_double conj (complex_double a, complex_double b)
complex_fract16 conj_fr16 (complex_fract16 a, complex_fract16 b)
```

Description

This function conjugates the complex input *a* and returns the result.

Algorithm

$$\begin{aligned}\operatorname{Re}(c) &= \operatorname{Re}(a) \\ \operatorname{Im}(c) &= -\operatorname{Im}(a)\end{aligned}$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for `conjf( )`, `conj( )`

-1.0 to 1.0 for `conj_fr16( )`

convolve

convolution

Synopsis

```
#include <filter.h>
void convolve_fr16(cin1, clen1, cin2, clen2, cout)
const fract16 cin1[];    /* pointer to input sequence 1 */
int clen1;               /* length of the input sequence 1 */
const fract16 cin2[];    /* pointer to input sequence 2 */
int clen2;               /* length of the input sequence 2 */
float cout[];            /* pointer to output sequence */
```

Description

This function convolves two sequences pointed to by `cin1` and `cin2`. If `cin1` points to the sequence whose length is `clen1` and `cin2` points to the sequence whose length is `clen2`, then resulting sequence pointed to by `cout` has length: `clen1 + clen2 - 1`

Algorithm

Convolution between two sequences `cin1` and `cin2` is described as:

$$cout[n] = \sum_{k=0}^{clen2-1} cin1[n+k-(clen2-1)] \bullet cin2[(clen2-1)-k]$$

for `n=0` to `clen1 + clen2 - 1`. Values for `cin1[j]` are considered to be zero for `j < 0` or `j > clen1 - 1`.

DSP Run-Time Library Reference

Example

Here is an example of a convolution where `cin1` is of length 4 and `cin2` is of length 3. If we represent `cin1` as “A” and `cin2` as “B”, the elements of the output vector are:

$\{A[0]*B[0],$

$A[1]*B[0] + A[0]*B[1],$

$A[2]*B[0] + A[1]*B[1] + A[0]*B[2],$

$A[3]*B[0] + A[2]*B[1] + A[1]*B[2],$

$A[3]*B[1] + A[2]*B[2],$

$A[3]*B[2]\}$

Domain

-1.0 to +1.0

conv2d

2-D convolution

Synopsis

```
#include <filter.h>
void conv2d_fr16(min1, mrow1, mcol1, min2, mrow2, mcol2, mout )
const fract16 *min1;      /* pointer to input matrix 1 */
int mrow1;                /* number of rows in matrix 1 */
int mcol1;                /* number of columns in matrix 1 */
const fract16 *min2;      /* pointer to input matrix 2 */
fract16 *mrow2;           /* number of rows in matrix 1 */
int mcol2;                /* number of columns in matrix 2 */
int *mout;                /* pointer to output matrix */
```

Description

This function computes 2-Dimensional Convolution of input matrix `min1` of size `mrow1` x `mcol1` and `min2` of size `mrow2` x `mcol2` and store the result in matrix `mout` of dimension $(mrow1 + mrow2 - 1) \times (mcol1 + mcol2 - 1)$.

Algorithm

Two dimensional input matrix `min1` is convolved with input matrix `min2`, placing the result in a matrix pointed to by `mout`.

$$mout[c, r] = \sum_{i=0}^{mcol2-1} \sum_{j=0}^{mrow2-1} min1[c+i, r+j] \bullet min2[(mcol2-1)-i, (mrow2-1)-j]$$

for `c = 0` to `mcol1+mcol2-1` and `r = 0` to `mrow2-1`.

Domain

-1.0 to +1.0

conv2d3x3

2-D convolution

Synopsis

```
#include <filter.h>
void conv2d3x3_fr16(min1, mrow1, mcol1, min2, mout )
const fract16 *min1[];    /* pointer to input matrix 1 */
int mrow1;                /* number of rows in matrix 1 */
int mcol1;                /* number of columns in matrix 1 */
const fract16 *min2[];    /* pointer to input matrix 2 */
int mout[];               /* pointer to output matrix */
```

Description

This function computes 2-Dimensional Convolution of matrix `min1` (size `mrow1 x mcol1`) with matrix `min2` (size 3x3).

Algorithm

Two dimensional input matrix `min1` is convolved with input matrix `min2`, placing the result in a matrix pointed to by `mout`.

$$mout [c, r] = \sum_{i=0}^2 \sum_{j=0}^2 min1 [c + i, r + j] \bullet min2 [2 - i, 2 - j]$$

for `c = 0` to `mcol1+2` and `r = 0` to `mrow1+2`.

Domain

-1.0 to +1.0

copysign

copysign

Synopsis

```
#include <math.h>
float copysignf (float parm1, float parm2)
double copysign (double parm1, double parm2)
fract16 copysign_fr16 (fract16 parm1, fract16 parm2)
```

Description

Copies the sign of the second argument to the first argument.

Algorithm

```
return(|parm1| * copysignof(parm2))
```

Domain

Full Range for type of parameters used.

DSP Run-Time Library Reference

countones

count one bits in word

Synopsis

```
#include <math.h>
int countones(int word)
int lcountones(long word)
```

Description

This function counts number of one bits in word.

Algorithm

return = sum(j=0 to word size -1) bit[j] of word

crosscoh

cross-coherence

Synopsis

```
#include <stats.h>
void crosscohf(a,b,n,m,c)
const float a[];          /* Input vector a */
const float b[];          /* Input vector b */
int n;                    /* Number of input samples */
int m;                    /* Lag count */
float c[];                /* Output vector c */
void crosscoh_fr16(a,n,m,c)
const fract16 a[];        /* Input vector a */
const fract16 b[];        /* Input vector b */
int n;                    /* Number of input samples */
int m;                    /* Lag count */
fract16 c[];              /* Output vector c */
```

Description

This function computes the cross-coherence of the input elements contained within input vector *a* and input vector *b*, and stores the result to output vector *c*.

Algorithm

$$c_k = \frac{1}{n} * \left(\sum_{j=0}^{n-k-1} (a_j - \bar{a}) * (b_{j+k} - \bar{b}) \right)$$

where $k=\{0,1,...,m-1\}$, $\bar{a}$ is the mean value of input vector *a* and $\bar{b}$ is the mean value of input vector *b*.

Domain

-3.4 x 10³⁸ to +3.4 x 10³⁸ for crosscohf()

-1.0 to +1.0 for crosscoh_fr16()

crosscorr

cross-correlation

Synopsis

```
#include <stats.h>
void crosscorr(a,b,n,m,c)
const float a[];          /* Input vector a */
const float b[];          /* Input vector b */
int n;                    /* Number of input samples */
int m;                    /* Lag count */
float c[];                /* Pointer to output vector c */
void crosscorr_fr16(a,b,n,m,c)
const fract16 a[];        /* Input vector a */
const fract16 b[];        /* Input vector b */
int n;                    /* Number of input samples */
int m;                    /* Lag count */
fract16 c[];              /* Pointer to output vector c */
```

Description

This function computes the cross-correlation of the input elements contained within input vector *a* and input vector *b*, and stores the result to output vector *c*.

Algorithm

$$c_k = \frac{1}{n} * \left(\sum_{j=0}^{n-k-1} a_j * b_{j+k} \right)$$

where $k=\{0,1,...,m-1\}$

Domain

-3.4 x 10³⁸ to +3.4 x 10³⁸ for crosscorr()

-1.0 to +1.0 for crosscorr_fr16()

csub

complex subtraction

Synopsis

```
#include <complex.h>
complex_float csubf (complex_float a, complex_float b)
complex_double csub (complex_double a, complex_double b)
complex_fract16 csub_fr16 (complex_fract16 a, complex_fract16 b)
```

Description

This function computes the complex subtraction of two complex inputs: a , and b , and return the result.

Algorithm

$$\begin{aligned}\operatorname{Re}(c) &= \operatorname{Re}(a) - \operatorname{Re}(b) \\ \operatorname{Im}(c) &= \operatorname{Im}(a) - \operatorname{Im}(b)\end{aligned}$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for `csubf( )`, `csub( )`

-1.0 to 1.0 for `csub_fr16( )`

DSP Run-Time Library Reference

fir

finite impulse response filter

Synopsis

```
#include <filter.h>

void fir_fr16(x,y,n,s)
    const fract16 x[]; /* Input sample vector x */
    fract16 y[];       /* Output sample vector y */
    int n;             /* Number of input samples */
    fir_state *s       /* Pointer to filter state structure */
```

The FIR filter function uses the following structure to maintain the state of the filter:

```
typedef struct
{
    fract16 *h,      /* filter coefficients */
    fract16 *d,      /* start of delay line */
    fract16 *p,      /* read/write pointer */
    int k,           /* no. of coefficients */
    int l            /* interpolation/decimation index*/
} fir_state_fr16;

void fir_init_fr16(fir_state_fr16 *s, fract16[], fract16 d[],
int k, int l);
```

Description

This function computes a FIR (Finite Impulse Response Filter) using the coefficients stored in vector `s->h` applied to the elements of vector `x`. The results are stored in vector `y`.

For efficiency, this filter uses a separate variable of type `fir_state` to maintain the filter state. You must declare space for this variable, and initialize it prior to using the filter. An initialization function, `fir_init` is provided.

Algorithm

$$y(k) = \sum_{i=0}^{p-1} h(i) * x(k-i) \text{ for } k = 0, 1, \dots, n$$

Domain

-1.0 to +1.0

fir_decima

FIR decimation filter

Synopsis

```
#include <filter.h>
void fir_decima_fr16(x,y,ng,s)
const fract16 x[];    /* Input sample vector x */
fract16 y[];          /* Output sample vector y */
int ng;               /* Number of samples to generate*/
fir_state_fr16 *s     /* Pointer to filter state structure */
```

Description

This function computes a FIR (Finite Impulse Response Filter) using the coefficients stored in vector `s->h` applied to the elements of vector `x`. The results are stored in vector `y` and computed according to a decimation index `s->l`. The number of desired output filter samples to generate is specified by the parameter `ng`, while the size of vector `s->h` is `s->k`, the size of input vector `x` is `ng`.

See the description of FIR for information about the `fir_state_fr16` structure and how to initialize it.

Algorithm

$$y(k) = \sum_{i=0}^{p-1} x(k * l - i) * h(i)$$

Domain

-1.0 to + 1.0

fir_interp

FIR interpolation filter

Synopsis

```
#include <filter.h>
void fir_interp_fr16(x,y,n,s)
const fract16 x[];      /* Input sample vector x */
fract16 y[];            /* Output sample vector y */
int n;                  /* Number of input samples */
fir_state_fr16 *s       /* Pointer to filter state structure */
```

Description

This function computes a FIR (Finite Impulse Response Filter) using the coefficients stored in vector `s.h` applied to the elements of vector `x`. The results are stored in vector `y` and computed according to a interpolation index, `s.l`. The number of input samples `x` is specified by the parameter `n`, the size of vector `s.h` is specified by `s.k`, and `s.p` and `s.l` must be integers, and the size of output vector `y` is `n`.

See the description of FIR for information about the `fir_state_fr16` structure and how to initialize it

Algorithm

$$y_m(k) = \sum_{i=0}^{p/l-1} x(k-i) * h(i * l + m)$$

where $m=\{0,1,2,...,l\}$

Domain

-1.0 to +1.0

gen_bartlett

generate bartlett window

Synopsis

```
#include <window.h>
void gen_bartlett_fr16(w,a,N)
fract16 w[]; /* Window vector */
int a;      /* Address stride in samples for window vector */
int N;      /* Length of window vector */
```

Description

This function generates a vector containing the Bartlett window. The length is specified by parameter N.



This window is similar to the Triangle window but has the following different properties [2]:

- The Bartlett window always returns a window with two zeros on either end of the sequence, so that for odd n, the center section of a N+2 Bartlett window equals a N Triangle window.
- For even n, the Bartlett window is still the convolution of two rectangular sequences. There is no standard definition for the Triangle window for even n; the slopes of the Triangle window are slightly steeper than those of the Bartlett window.

Algorithm

$$w[n] = 1 - \left| \frac{n - \frac{N-1}{2}}{\frac{N-1}{2}} \right|$$

where $n = \{0, 1, 2, \dots, N-1\}$

Domain

$a > 0; N > 0$

gen_blackman

generate blackman window

Synopsis

```
#include <window.h>
void gen_blackman_fr16(w,a,N)
fract16 w[];      /* Window vector */
int a;            /* Address stride in samples for window vector */
int N;            /* Length of window vector */
```

Description

This function generates a vector containing the Blackman window. The length is specified by parameter N.

Algorithm

$$w[n] = 0.42 - 0.5 \cos\left(\frac{2\pi n}{N-1}\right) + 0.08 \cos\left(\frac{4\pi n}{N-1}\right)$$

where $n = \{0, 1, 2, \dots, N-1\}$

Domain

$a > 0$; $N > 0$

gen_gaussian

generate gaussian window

Synopsis

```
#include <window.h>
void gen_gaussian_fr16(w,alpha,a,N)
fract16 w[]; /* Window vector */
float alpha; /* Gaussian alpha parameter */
int a;       /* Address stride in samples for window vector */
int N;       /* Length of window vector */
```

Description

This function generates a vector containing the Gaussian window. The length is specified by parameter N.

Algorithm

$$w(n) = \exp \left[-\frac{1}{2} \left(\alpha \frac{n - N/2 - 1/2}{N/2} \right)^2 \right]$$

where $n = \{0, 1, 2, \dots, N-1\}$ and is an input parameter

Domain

$a > 0$; $N > 0$; $\alpha > 0.0$

gen_hamming

generate hamming window

Synopsis

```
#include <window.h>
void gen_hamming_fr16(w,a,N)
fract16 w[]; /* Window vector */
int a;      /* Address stride in samples for window vector */
int N;      /* Length of window vector */
```

Description

This function generates a vector containing the Hamming window. The length is specified by parameter N.

Algorithm

$$w[n] = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N-1}\right)$$

where $n = \{0, 1, 2, \dots, N-1\}$

Domain

$a > 0$; $N > 0$

gen_hanning

generate hanning window

Synopsis

```
#include <window.h>
void gen_hanning_fr16(w,a,N)
fract16 w[]; /* Window vector */
int a;      /* Address stride in samples for window vector */
int N;      /* Length of window vector */
```

Description

This function generates a vector containing the Hanning window. The length is specified by parameter N. This window is also known as the Cosine window.

Algorithm

$$w[n] = 0.5 - 0.5 \cos\left(\frac{2\pi n}{N+1}\right)$$

where $n = \{1, 1, 2, \dots, N+1\}$

Domain

$a > 0$; $N > 0$

gen_harris

generate harris window

Synopsis

```
#include <window.h>
void gen_harris_fr16(w,a,N)
fract16 w[]; /* Window vector */
int a;      /* Address stride in samples for window vector */
int N;      /* Length of window vector */
```

Description

This function generates a vector containing the Harris window. The length is specified by parameter N. This window is also known as the Blackman-Harris window.

Algorithm

$$w[n] = 0.35875 - 0.48829 * \cos\left(\frac{2\pi n}{N-1}\right) + 0.14128 * \cos\left(\frac{4\pi n}{N-1}\right) + 0.01168 * \cos\left(\frac{6\pi n}{N-1}\right)$$

where $n = \{0, 1, 2, \dots, N-1\}$

Domain

$a > 0$; $N > 0$

gen_kaiser

generate kaiser window

Synopsis

```
#include <window.h>
void gen_kaiser_fr16(w,beta,a,N)
fract16 w[]; /* Window vector */
float beta; /* Kaiser beta parameter */
int a; /* Address stride in samples for window vector */
int N; /* Length of window vector */
```

Description

This function generates a vector containing the Kaiser window. The length is specified by parameter N. The β value is specified by parameter b. Refer to [3] for details on obtaining the value of β .

Algorithm

$$w[n] = \frac{I_0 \left[\beta \left(1 - \left[\frac{n - \alpha}{\alpha} \right]^2 \right)^{1/2} \right]}{I_0(\beta)}$$

where $n = \{0, 1, 2, \dots, N-1\}$, $\alpha = (N - 1) / 2$, and $I_0(\cdot)$ represents the zeroth-order modified Bessel function of the first kind.

Domain

$a > 0$; $N > 0$; $\alpha > 0.0$

gen_rectangular

generate rectangular window

Synopsis

```
#include <window.h>
void gen_rectangular_fr16(w,a,N)
fract16 w[]; /* Window vector */
int a;      /* Address stride in samples for window vector */
int N;      /* Length of window vector */
```

Description

This function generates a vector containing the rectangular window. The length is specified by parameter N.

Algorithm

$w[n] = 1$ where $n = \{0, 1, 2, \dots, N-1\}$

Domain

$a > 0$; $N > 0$

gen_triangle

generate triangle window

Synopsis

```
#include <window.h>
void gen_triangle_fr16(w,a,N)
fract16 w[]; /* Window vector */
int a;      /* Address stride in samples for window vector */
int N;      /* Length of window vector */
```

Description

This function generates a vector containing the Triangle window. The length is specified by parameter *N*. Refer to the Bartlett window regarding the relationship between it and the Triangle window.

Algorithm

For even *n* the following equation applies:

$$w[n] = \begin{cases} \frac{2n+1}{N} & n < N/2 \\ \frac{2N-2n-1}{N} & n > N/2 \end{cases}$$

where $n = \{0, 1, 2, \dots, N-1\}$

For odd *n* the following equation applies:

$$w[n] = \begin{cases} \frac{2n+2}{N+1} & n < N/2 \\ \frac{2N-2n}{N+1} & n > N/2 \end{cases}$$

where $n = \{0, 1, 2, \dots, N-1\}$

Domain

$a > 0$; $N > 0$

gen_vonhann

generate von hann window

Synopsis

```
#include <window.h>
void gen_vonhann_fr16(w,a,N)
fract16 w[]; /* Window vector */
int a;       /* Address stride in samples for window vector */
int N;       /* Length of window vector */
```

Description

This function is identical to gen_hanning window.

Domain

$a > 0$; $N > 0$

histogram

histogram

Synopsis

```
#include <stats.h>
void histogramf(a,c,max,min,n,m)
const float a[];      /* Pointer to input vector a */
int c[];              /* Pointer to output vector c */
float max;            /* Maximum value of the bin */
float min;            /* Minimum value of the bin */
int n;                /* Number of input samples */
int m;                /* Number of bins */

void histogram_fr16(a,c,max,min,n,m)
const fract16 a[];    /* Pointer to input vector a */
int c[];              /* Pointer to output vector c */
fract16 max;          /* Maximum value of the bin */
fract16 min;          /* Minimum value of the bin */
int n;                /* Number of input samples */
int m;                /* Number of bins */
```

Description

This function computes the histogram of the input elements contained within input vector *a*, and stores the result to output vector *c*.

Algorithm

It bins the element of input vector *x* into *m* equally spaced containers, and returns the number of elements in each container.

Domain

-3.4 x 10³⁸ to +3.4 x 10³⁸ for histogramf()

-1.0 to +1.0 for histogram_fr16()

ifft

N point inverse FFT

Synopsis

```
#include <filter.h>
void ifft_fr16(in[], t[], out[], w[], wst, n, block_exponent,
scale_mc)
const complex_fract16 in[]; /* input sequence */
complex_fract16 t[];        /* temporary working buffer */
complex_fract16 out[];      /* output sequence */
const complex_fract16 w[];  /* twiddle sequence */
int wst;                    /* twiddle factor stride */
int n;                      /* number of FFT points */
int block_exponent;         /* block exponent of output data */
int scale_method;           /* scaling method desired 0-none,
                             1-static, 2-dynamic */
```

Description

This function transforms the frequency domain complex input signal sequence to the time domain by using the radix-2 'Inverse Fast Fourier Transform'. If input data can be overwritten, the optimum memory usage is achieved by setting the output pointer to the input array.

For efficiency, the “twiddle table” is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

If the twiddle table has been allocated at a larger size than needed for a particular call of `iffttrad4`, then the stride parameter will need to be set appropriately; otherwise, it should be one. [For more information, see “twidfft” on page 4-90.](#)

Algorithm

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-nk}$$

The implementation uses core FFT functions. To get the inverse effect, it first swaps the real and imaginary parts of the input, performs the direct radix-2 transformation and finally swaps the real and imaginary parts of the output.

Domain

Input sequence length n must be a power of two and at least 16.

iffttrad4

N point inverse FFT

Synopsis

```
#include <filter.h>
void iffttrad4_fr16 (in[], t[], out[], w[], wst, n,
block_exponent, scale_mc)
const complex_fract16 in[]; /* input sequence */
complex_fract16 t[];        /* temporary working buffer */
complex_fract16 out[];      /* output sequence */
const complex_fract16 w[];  /* twiddle sequence */
int wst;                    /* twiddle factor stride */
int n;                      /* number of FFT points */
int block_exponent;         /* block exponent of output data */
int scale_method;           /* scaling method desired 0-none,
                             1-static, 2-dynamic */
```

Description

This function transforms the frequency domain complex input signal sequence to the time domain by using the radix-4 'Inverse Fast Fourier Transform'. If input data can be overwritten, the optimum memory usage is achieved by setting the output pointer to the input array.

For efficiency, the “twiddle table” is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

If the twiddle table has been allocated at a larger size than needed for a particular call of `ifft`, then the stride parameter will need to be set appropriately; otherwise, it should be one. [For more information, see “twidfft” on page 4-90.](#)

Algorithm

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-nk}$$

The implementation uses core FFT functions implemented as direct radix-4 algorithm. To get the inverse effect, it first swaps the real and imaginary parts of the input, performs the direct radix-4 transformation and finally swaps the real and imaginary parts of the output.

Domain

Input sequence length n must be a power of four and at least 16.

ifft2d

NxN point 2-D inverse input FFT

Synopsis

```
#include <filter.h>
void ifft2d_fr16(*in, *t, *out, w[], wst, n, block_exponent,
scale_mc)
const complex_float *in; /* pointer to input matrix a[n][n] */
complex_fract16 *t; /* pointer to working buffer
t[n][n] */
complex_fract16 *out; /* pointer to output matrix c[n][n] */
const complex_fract16 w[]; /* twiddle sequence */
int wst; /* twiddle factor stride */
int n; /* number of FFT points */
int block_exponent; /* block exponent of output data */
int scale_method; /* scaling method desired 0-none,
1-static, 2-dynamic */
```

Description

This function computes two dimensional Inverse Fast Fourier Transform of the complex input matrix $a[n][n]$, and stores the result to the complex output matrix $c[n][n]$.

If input data can be overwritten, the optimum memory usage is achieved by setting the output pointer to the input array.

For efficiency, the “twiddle table” is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

If the twiddle table has been allocated at a larger size than needed for a particular call of `ifft2d`, then the stride parameter will need to be set appropriately; otherwise, it should be one. [For more information, see “twidfft” on page 4-90.](#)

Algorithm

$$c(i, j) = \frac{1}{n^2} \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} a(k, l) * e^{2\pi j(i*k + j*l)/n}$$

where $i=\{0,1,\dots,n-1\}$, $j=\{0,1,2,\dots,n-1\}$

Domain

Input sequence length n must be a power of two and at least 16.

iir

infinite impulse response filter

Synopsis

```
#include <filter.h>
void iir_fr16(x,y,n,s)
const fract16 x[];      /* Input sample vector x */
fract16 y[];             /* Output sample vector y */
int n;                  /* Number of input samples */
iir_state_fr16 *s        /* Pointer to filter state structure */
```

The IIR filter function uses the following structure to maintain the state of the filter:

```
typedef struct
{
    float *c,           /* coefficients */
    float *d,           /* start of delay line */
    int k               /* no. of bi-quad stages */
} iir_state_fr16;
```

These structure can be initialized with the following function/macro:

```
void iir_init_fr16(iir_state_fr16 *s, float c[], float d[], int k);
```

Description

Using a bi-quad implementation, this function computes an IIR (Infinite Impulse Response Filter) using the coefficients stored in vector `s->c`, delay node points stored in input buffer `s->d`, and applied to the elements of vector `x`. The results are stored in vector `y`.

For efficiency, this filter uses a separate variable of type `iir_state_fr16` to maintain the filter state. You must declare space for this variable, and initialize it prior to using the filter. An initialization function, `iir_init_fr16` is provided.

Algorithm

$$H(z) = \frac{B_0 + B_1 z^{-1} + B_2 z^{-2}}{1 - A_1 z^{-1} - A_2 z^{-2}}$$

where

$$\begin{aligned} D_m &= A_2 * D_{m-2} + A_1 * D_{m-1} + x_m \\ Y_m &= B_2 * D_{m-2} + B_1 * D_{m-1} + B_0 * D_m \end{aligned}$$

where $m=\{0,1,2,\dots,n-1\}$

Domain

-1.0 to +1.0

Notes

The B and A coefficients are passed through the `s.c` vector in the `iir_state_fr16` structure. `s->c[n+0] = A2`, `s->c[n+1] = A1`, `s->c[n+2] = B2`, `s->c[n+3] = B1`, `s->c[n+4] = B0`. The value of A0 is implied to be 1.0 and A1 and A2 must be scaled accordingly.

DSP Run-Time Library Reference

max

maximum

Synopsis

```
#include <math.h>
int max (int parm1, int parm2)
float fmaxf (float parm1, float parm2)
double fmax (double parm1, double parm2)
fract16 max_fr16 (fract16 parm1, fract16 parm2)
```

Description

Return the larger of its 2 arguments.

Algorithm

```
If (parm1 > parm2)
return(parm1)
else
return(parm2)
```

Domain

Full Range for type of parameters.

mean

mean

Synopsis

```
#include <stats.h>
fract16 mean_fr16(a,n)
const fract16 a[]; /* Input vector a */
int n;             /* Number of input samples */
float meanf(a,n)
const float a[];   /* Input vector a */
int n;             /* Number of input samples */
```

Description

This function computes the mean of the input elements contained within input vector *a* and return the result.

Algorithm

$$c = \frac{1}{n} * \left(\sum_{i=0}^{n-1} a_i \right)$$

Domain

-3.4 x 10³⁸ to +3.4 x 10³⁸ for meanf()

-1.0 to +1.0 for mean_fr16()

DSP Run-Time Library Reference

min

minimum

Synopsis

```
#include <math.h>
int min (int parm1, int parm2)
float fminf (float parm1, float parm2)
double fmin (double parm1, double parm2)
fract16 min_fr16 (fract16 parm1, fract16 parm2)
```

Description

Return the smaller of its 2 arguments.

Algorithm

```
If (parm1 < parm2)
return(parm1)
else
return(parm2)
```

Domain

Full Range for type of parameters used.

mu_compress

μ-law compression

Synopsis

```
#include <filter.h>
void mu_compress(in, out, n)
const short in[];    /* Input array */
short out[];         /* Output array */
int n;               /* number of elements to be compressed */
```

Description

This function performs μ-law compression (ITU rec. G.711) on the elements of the input vector pointed to by `in` and outputs compressed data in vector pointed to by `out`.

Algorithm

$C(k) = \text{mu_law compression of } A(k) \text{ for } k=0 \text{ to } n-1$

Domain

content of input array: -8192 to 8191

mu_expand

μ-law expansion

Synopsis

```
#include <filter.h>
void mu_expand(in, out, n)
const short in[];      /* Input array */
short out[];           /* Output array */
int n;                 /* number of elements to be expanded */
```

Description

This function performs μ-law expansion (ITU rec. G.711) on the elements of the input vector pointed to by `in` and outputs expanded data in vector pointed to by `out`.

Algorithm

$C(k) = \text{mu_law expansion of } A(k) \text{ for } k=0 \text{ to } n-1$

Domain

content of input array: 0 to 255

norm

normalization

Synopsis

```
#include <complex.h>
complex_float normf (complex_float a)
complex_double norm (complex_double a)
```

Description

This function normalizes the complex input *a* and returns the result.

Algorithm

$$\text{Re}(c) = \frac{\text{Re}(a)}{\sqrt{\text{Re}^2(a) + \text{Im}^2(a)}}$$

$$\text{Im}(c) = \frac{\text{Im}(a)}{\sqrt{\text{Re}^2(a) + \text{Im}^2(a)}}$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

polar

construct from polar coordinates

Synopsis

```
#include <complex.h>
complex_float polarf (complex_float mag, complex_float phase)
complex_double polar (complex_double mag, complex_double phase)
complex_fract16 polar_fr16 (complex_fract16 mag, complex_fract16
                             phase)
```

Description

This function transforms the polar coordinate to normal coordinate.

Algorithm

$$\text{Re}(c) = r \cdot \cos(\theta)$$

$$\text{Im}(c) = r \cdot \sin(\theta)$$

Domain

phase = [-9099 ... 9099] for polarf(), polar()

mag = -3.4×10^{38} to $+3.4 \times 10^{38}$ for polarf(), polar()

phase = -1.0 to +1.0 for polar_fr16()

mag = -1.0 to +1.0 for polar_fr16()

rfft

N point real input FFT

Synopsis

```
#include <filter.h>
void rfft_fr16(in[], t[], out[], w[], wst, n, block_exponent,
scale_mc)
const fract16 in[];          /* input/output sequence */
complex_fract16 t[];         /* temporary working buffer */
complex_fract16 out[];       /* working buffer */
const complex_fract16 w[];   /* twiddle sequence */
int wst;                     /* twiddle factor stride */
int n;                       /* number of FFT points */
int block_exponent;         /* block exponent of output data */
int scale_method;           /* scaling method desired 0-none,
                             1-static, 2-dynamic */
```

Description

This function transforms the time domain real input signal sequence to the frequency domain by using the radix-2 'Fast Fourier Transform'. RFFT takes advantage of the fact that the imaginary part of the input equals zero, which in turn eliminates half of the multiplications in the butterfly.

If input data can be overwritten, the optimum memory usage is achieved by setting the output pointer to the input array and providing that the input array is 2N.

For efficiency, the “twiddle table” is calculated once, during initialization, and then provided to the RFFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

DSP Run-Time Library Reference

If the twiddle table has been allocated at a larger size than needed for a particular call of `rfft`, then the stride parameter will need to be set appropriately; otherwise, it should be one. [For more information, see “twidfft” on page 4-90.](#)

Algorithm

See ‘N Point Complex Input FFT’. Output sequence will have $n/2$ complex members due to a fact that fft of real input data produces symmetric output around half sampling frequency point.

Domain

Input sequence length n must be a power of two and at least 16.

rfftrad4

N point real input FFT

Synopsis

```

#include <filter.h>
void rfftrad4_fr16(in[], t[], out[], w[], wst, n,
block_exponent, scale_mc)
void _rfft(in[], t[], out[], w[], wst, n)
const fract16 in[];          /* input/output sequence */
complex_fract16 t[];         /* temporary working buffer */
complex_fract16 out[];       /* working buffer */
const complex_fract16 w[];   /* twiddle sequence */
int wst;                     /* twiddle factor stride */
int n;                       /* number of FFT points */
int block_exponent;         /* block exponent of output data */
int scale_method;           /* scaling method desired 0-none,
                             1-static, 2-dynamic */

```

Description

This function transforms the time domain real input signal sequence to the frequency domain by using the radix-4 'Fast Fourier Transform'. RFFT takes advantage of the fact that the imaginary part of the input equals zero, which in turn eliminates half of the multiplications in the butterfly.

If input data can be overwritten, the optimum memory usage is achieved by setting the output pointer to the input array and providing that the input array is 2N.

For efficiency, the “twiddle table” is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

DSP Run-Time Library Reference

If the twiddle table has been allocated at a larger size than needed for a particular call of `rfft`, then the stride parameter will need to be set appropriately; otherwise, it should be one. [For more information, see “twidfft” on page 4-90.](#)

Algorithm

See ‘N Point Complex Input FFT’. Output sequence will have $n/2$ complex members due to a fact that fft of real input data produces symmetric output around half sampling frequency point.

Domain

Input sequence length n must be a power of four and at least 16.

rfft2d

NxN point 2-D real input FFT

Synopsis

```
#include <filter.h>
```

```
void rfft2d_fr16(*in, *t, *out, w[], wst, n, block_exponent,
scale_mc)
const fract16 *in;          /* pointer to input matrix a[n][n] */
complex_fract16 *t;         /* pointer to working buffer t[n][n] */
complex_fract16 *out;       /* pointer to output matrix [n][n] */
const complex_fract16 w[]; /* twiddle sequence */
int wst;                    /* twiddle factor stride */
int n;                      /* number of FFT points */
int block_exponent;        /* block exponent of output data */
int scale_method;          /* scaling method desired 0-none,
                           1-static, 2-dynamic */
```

Description

This function computes a two dimensional Fast Fourier Transform of the real input matrix $a[n][n]$, and stores the result to the complex output matrix $c[n][n]$.

If input data can be overwritten, the optimum memory usage is achieved by setting the output pointer to the input array and providing that the input array is 2 times NxN.

For efficiency, the “twiddle table” is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

DSP Run-Time Library Reference

If the twiddle table has been allocated at a larger size than needed for a particular call of `rfft2d`, then the stride parameter will need to be set appropriately; otherwise, it should be one. [For more information, see “twidfft” on page 4-90.](#)

Algorithm

$$c(i, j) = \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} a(k, l) * e^{-2\pi j(i*k + j*l)/n}$$

where $i=\{0,1,\dots,n-1\}$, $j=\{0,1,2,\dots,n-1\}$

Domain

Input sequence length n must be a power of two and at least 16.

rms

root mean square

Synopsis

```
#include <stats.h>
float rmsf(a,n)
const float a[];      /* Pointer to input vector a */
int n;                /* number of input samples */
fractl6 rms_fr16(a,n)
const fractl6 a[];    /* Pointer to input vector a */
int n;                /* Number of input samples */
```

Description

This function computes the root mean square of the input elements contained within input vector *a* and return the result.

Algorithm

$$c = \sqrt{\frac{\sum_{i=0}^{n-1} a_i^2}{n}}$$

Domain

-3.4 x 10³⁸ to +3.4 x 10³⁸ for rmsf()

-1.0 to +1.0 for rms_fr16()

twidfft

generate FFT twiddle factors

Synopsis

```
#include <filter.h>
void twidfft(w[], n)
complex_fract16 w[];    /* twiddle sequence */
int n;                  /* number of FFT points */
```

Description

Various different versions of the FFT are provided, including `cfft`, `rfft`, `ifft`, `cfft2d`, `rfft2d`, `ifft2d`. The number of points is provided as an argument; both radix 2 or radix 4 implementations are provided.

For efficiency, the `twiddle table` is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the array and initialize it prior to calling an FFT function. The function `twidfft` is the initialization function. The size of the table must be 3/4 the largest FFT to be processed.

Several FFT's of different sizes can all be accommodated with the same twiddle factor table. Simply allocate the table at the maximum size. Each FFT has an additional parameter, the “stride” of the twiddle table. To use the whole table, specify a stride of 1. If your FFT uses only half the points of the largest, the stride should be 2 (this takes only every other element).

Algorithm

This function takes FFT length `n` as an input parameter and generates the lookup table of complex twiddle coefficients. 3/4 of one period of sine/cosine is described by $3/4 n$ complex samples in the lookup table.

The samples are generated as follows:

$$twid_re(k) = \cos\left(\frac{2\pi}{n}k\right)$$

$$twid_im(k) = \sin\left(\frac{2\pi}{n}k\right)$$

for k=0 to 3/4 n

Domain

n must be a power of two and at least 16.

DSP Run-Time Library Reference

var

variance

Synopsis

```
#include <stats.h>
float varf(a,n)
const float a[];      /* Pointer to input vector a */
int n;                /* number of input samples */
fract16 var_fr16(a, n)
const fract16 a[];    /* Pointer to input vector a */
int n;                /* number of input samples */
```

Description

This function computes the variance of the input elements contained within input vector *a* and returns the result.

Algorithm

$$c = \frac{n * \sum_{i=0}^{n-1} a_i^2 - (\sum_{i=0}^{n-1} a_i)^2}{n(n-1)}$$

Domain

-3.4 x 10³⁸ to +3.4 x 10³⁸ for varf()

-1.0 to +1.0 for var_fr16()

zero_cross

count zero crossing

Synopsis

```
#include <stats.h>
int zero_crossf(a,n)
const float a[];          /* Pointer to input vector a */
int n;                    /* Number of input samples */
zero_cross_fr16 (a, n)
const fract16 a[];        /* Pointer to input vector a */
int n;                    /* Number of input samples */
```

Description

This function computes the number of times that a signal crosses over the zero line and returns the result.

Algorithm

The actual algorithm is different from what is shown below because we need to handle the case where an element of the array is zero, but this gives you an understanding;

```
if (a(i) > 0 && a(i+1) < 0) || (a(i) < 0 && a(i+1) > 0))
number of zeros increased by one
```

Domain

-3.4 x 10³⁸ to +3.4 x 10³⁸ for zero_crossf()

-1.0 to +1.0 for zero_cross_fr16()

DSP Run-Time Library Reference