4 DSP RUN-TIME LIBRARY

This chapter describes the DSP run-time library which contains a broad collection of functions that are commonly required by signal processing applications. The services provided by the DSP run-time library include support for general-purpose signal processing such as companders, filters, and Fast Fourier Transform (FFT) functions. All these services are Analog Devices extensions to ANSI standard C. These support functions are in addition to the C/C++ run-time library functions that are described in Chapter 3, "C/C++ Run-Time Library" (The library also contains functions called implicitly by the compiler, for example `div32`.)

For more information about the algorithms on which many of the DSP run-time library's math functions are based, see Cody, W. J. and W. Waite, *Software Manual for the Elementary Functions,* Englewood Cliffs, New Jersey: Prentice Hall, 1980.

In addition to containing the user-callable functions described in this chapter, the DSP run-time library also contains compiler support functions which perform basic operations on integer and floating-point types that the compiler might not perform in-line. These functions are called by compiler generated code to implement, for example, basic type conversions, floating-point operations, etc. Note that the compiler support functions should not be called directly from user code.

This chapter contains:

- "DSP Run-Time Library Guide" on page 4-2
  contains information about the library and provides a description of the DSP header files that are included with this release of the `ccblkfn` compiler.
- "DSP Run-Time Library Reference" on page 4-46
  contains the complete reference for each DSP run-time library function provided with this release of the `ccblkfn` 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 about:

- "Linking DSP Library Functions"
- "Working With Library Source Code" on page 4-4
- "Library Attributes" on page 4-4
- "DSP Header Files" on page 4-5
- "Measuring Cycle Counts" on page 4-36

Linking DSP Library Functions

The DSP run-time library is located under the VisualDSP++ installation directory in the subdirectory `Blackfin/lib`. Different versions of the library are supplied and catalogued in Table 4-1.

Versions of the DSP run-time library containing `532` in the library file-name have been built to run on any of the ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF534, ADSP-BF536, ADSP-BF537,

Table 4-1. DSP Library Files

Blackfin/lib Directory	Description
libdsp532.dlb libdsp535.dlb libdsp561.dlb	DSP run-time library
libdsp532y,dlb libdsp535y.dlb libdsp561y.dlb	DSP run-time library built with the -si-revision flag specified (For more information, see "-si-revision version" on page 1-64.).

ADSP-BF538, or ADSP-BF539 processors. Versions of the DSP run-time library containing 535 in the library filename have been built to run on any of of the ADSP-BF535, AD65xx, or AD69xx processors. Versions of the DSP run-time library containing 561 in the library filename have been built to run on either the ADSP-BF561 or ADSP-BF566 processors.

Versions of the library whose file name end with a y (for example, libdsp532y.dlb) have been built with the compiler's -si-revision switch and include all available compiler workarounds for hardware anomalies. (See "-si-revision version" on page 1-64.)

When an application calls a DSP library function, the call creates a reference that the linker resolves. One way to direct the linker to the library's location is to use the default Linker Description File (<your_target>.ldf). If a customized .ldf file is used to link the application, then the appropriate DSP run-time library should be added to the .ldf file used by the project.

ⓘ Instead of modifying a customized .ldf file, the -l switch (see "-l library" on page 1-40) can be used to specify which library should be searched by the linker. For example, the -ldsp532 switch adds the library libdsp532.dlb to the list of libraries that the linker examines. For more information on .ldf files, see the *VisualDSP++ 4.5 Linker and Utilities Manual.*

Working With Library Source Code

The source code for the functions in the DSP run-time library is provided with your VisualDSP++ software. By default, the libraries are installed in the directory `Blackfin/lib` and the source files are copied into the directory `Blackfin/lib/src`. Each function is kept in a separate file. The file name is the name of the function with `.asm` or `.c` extension. 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 specific functions can be customized as a user requires. To modify these files, proficiency in Blackfin assembly language and an understanding of the run-time environment is needed.

Refer to "C/C++ Run-Time Model and Environment" on page 1-281 for more information.

Before making any modifications to the source code, copy it to a file with a different file name 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.

Library Attributes

The DSP run-time library contains the same attributes as the C/C++ run-time library. For more information, see "Library Attributes" in Chapter 3, C/C++ Run-Time Library.

DSP Header Files

The DSP header files contains prototypes for all the DSP library functions. When the appropriate #include preprocessor command is included in your source, the compiler uses the prototypes to check that each function is called with the correct arguments. The DSP header files included in this release of the ccblkfn compiler are:

- "complex.h – Basic Complex Arithmetic Functions"
- "cycle_count.h – Basic Cycle Counting" on page 4-9
- "cycles.h – Cycle Counting with Statistics" on page 4-9
- "filter.h – Filters and Transformations" on page 4-9
- "math.h – Math Functions" on page 4-14
- "matrix.h – Matrix Functions" on page 4-17
- "stats.h – Statistical Functions" on page 4-24
- "vector.h – Vector Functions" on page 4-24
- "window.h – Window Generators" on page 4-27

complex.h – Basic Complex Arithmetic Functions

The complex.h header file contains type definitions and basic arithmetic operations for variables of type complex_float, complex_double, complex_long_double, and complex_fract16.

The complex functions defined in this header file are listed in Table 4-2 on page 4-7. All the functions that operate in the complex_fract16 data type will use saturating arithmetic.

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
{
   long double re;
   long double im;
} complex_long_double;

typedef struct
{
   fract16 re;
   fract16 im;
} complex_fract16;
```

Details of the basic complex arithmetic functions are included in "DSP Run-Time Library Reference" starting on page 4-46.

Table 4-2. Complex Functions

Description	Prototype

Complex Absolute Value	`double cabs (complex_double a)`
	`float cabsf (complex_float a)`
	`long double cabsd (complex_long_double 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_long_double caddd`
	`  (complex_long_double a, complex_long_double 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_long_double csubd`
	`  (complex_long_double a, complex_long_double b)`
	`complex_fract16 csub_fr16`
	`  (complex_fract16 a, complex_fract16 b)`

Complex Multiply	`complex_double cmlt`
	`  (complex_double a, complex_double b)`
	`complex_float cmltf`
	`  (complex_float a, complex_float b)`
	`complex_long_double cmltd`
	`  (complex_long_double a, complex_long_double b)`
	`complex_fract16 cmlt_fr16`
	`  (complex_fract16 a, complex_fract16 b)`

Complex Division	`complex_double cdiv`
	` (complex_double a, complex_double b)`
	`complex_float cdivf`
	`  (complex_float a, complex_float b)`
	`complex_long_double cdivd`
	`  (complex_long_double a, complex_long_double b)`
	`complex_fract16 cdiv_fr16`
	`  (complex_fract16 a, complex_fract16 b)`

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

Description	Prototype

Get Phase of a Complex Number	`double arg (complex_double a)`
	`float argf (complex_float a)`
	`long double argd (complex_long_double a)`
	`fract16 arg_fr16 (complex_fract16 a)`

Complex Conjugate	`complex_double conj (complex_double a)`
	`complex_float conjf (complex_float a)`
	`complex_long_double conjd (complex_long_double a)`
	`complex_fract16 conj_fr16 (complex_fract16 a)`

Convert Cartesian to Polar Coordinates	`double cartesian (complex_double a, double* phase)`
	`float cartesianf (complex_float a, float* phase)`
	`long double cartesiand (complex_long_double a, long_double* phase)`
	`fract16 cartesian_fr16 (complex_fract16 a, fract16* phase)`

Convert Polar to Cartesian Coordinates	`complex_double polar (double mag, double phase)`
	`complex_float polarf (float mag, float phase)`
	`complex_long_double polard (long double mag, long double phase)`
	`complex_fract16 polar_fr16 (fract16 mag, fract16 phase)`

Complex Exponential	`complex_double cexp (double a)`
	`complex_long_double cexpd (long double a)`
	`complex_float cexpf (float a)`

Normalization	`complex_double norm (complex_double a)`
	`complex_long_double normd (complex_long_double a)`
	`complex_float normf (complex_float a)`

cycle_count.h – Basic Cycle Counting

The `cycle_count.h` header file provides an inexpensive method for benchmarking C-written source by defining basic facilities for measuring cycle counts. The facilities provided are based upon two macros, and a data type which are described in more detail in the section "Measuring Cycle Counts" on page 4-36.

cycles.h – Cycle Counting with Statistics

The `cycles.h` header file defines a set of five macros and an associated data type that may be used to measure the cycle counts used by a section of C-written source. The macros can record how many times a particular piece of code has been executed and also the minimum, average, and maximum number of cycles used. The facilities that are available via this header file are described in the section "Measuring Cycle Counts" on page 4-36.

filter.h – Filters and Transformations

The `filter.h` header file contains filters used in signal processing. It also includes the A-law and µ-law companders that are used by voice-band compression and expansion applications.

This header file also contains functions that perform key signal processing transformations, including FFTs and convolution.

Various forms of the FFT function are provided by the library corresponding to radix-2, radix-4, and two-dimensional FFTs. The number of points is provided as an argument. The header file also defines a complex FFT function that has been implemented using an optimized radix-4 algorithm. However, this function, `cfftf_fr16`, has certain requirements that may not be appropriate for some applications. The twiddle table for the FFT functions is supplied as a separate argument and is normally calculated once during program initialization.

The `cfftf_fr16` library function makes use of the `M3` register. The `M3` register may be used by an emulator for context switching. Refer to the appropriate emulator documentation.

Library functions are provided to initialize a twiddle table. A table can accommodate several FFTs of different sizes by allocating the table at maximum size, and then using the stride argument of the FFT function to specify the step size through the table. If the stride argument is set to 1, the FFT function uses all the table; if the FFT uses only half the number of points of the largest, the stride is 2.

The functions defined in this header file are listed in Table 4-3 and Table 4-4 and are described in detail in "DSP Run-Time Library Reference" on page 4-46.

Table 4-3. Filter Library

Description	Prototype
Finite Impulse Response Filter	`void fir_fr16 (const fract16 input[], fract16 output[], int length, fir_state_fr16 *filter_state)`
Infinite Impulse Response Filter	`void iir_fr16 (const fract16 input[], fract16 output[], int length, iir_state_fr16 *filter_state)`
Direct Form I Infinite Response Filter	`void iirdf1_fr16 (const fract16 input[], fract16 output[], int length, iirdf1_fr16_state *filter_state)`
FIR Decimation Filter	`void fir_decima_fr16 (const fract16 input[], fract16 output[], int length, fir_state_fr16 *filter_state)`

Table 4-3. Filter Library (Cont'd)

Description	Prototype
FIR Interpolation Filter	`void fir_interp_fr16 (const fract16 input[], fract16 output[], int length, fir_state_fr16 *filter_state)`
Complex Finite Impulse Response Filter	`void cfir_fr16 (const complex_fract16 input[], complex_fract16 output[], int length, cfir_state_fr16 *filter_state)`
Convert Coefficients for DF1 IIR	`void coeff_iirdf1_fr16 (const float acoeff[], const float bcoeff[ ], fract16 coeff[], int nstages)`

Table 4-4. Transformational Functions

Description	Prototype
Fast Fourier Transforms	
Generate FFT Twiddle Factors	`void twidfft_fr16 (complex_fract16 twiddle_table[], int fft_size)`
Generate FFT Twiddle Factors for Radix-2 FFT	`void twidfftrad2_fr16 (complex_fract16 twiddle_table[], int fft_size)`
Generate FFT Twiddle Factors for Radix-4 FFT	`void twidfftrad4_fr16 (complex_fract16 twiddle_table[], int fft_size)`
Generate FFT Twiddle Factors for 2-D FFT	`void twidfft2d_fr16 (complex_fract16 twiddle_table[], int fft_size)`
Generate FFT Twiddle Factors for Optimized FFT	`void twidfftf_fr16 (complex_fract16 twiddle_table[], int fft_size)`

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

Description	Prototype
N Point Radix-2 Complex Input FFT	void cfft_fr16 (const complex_fract16 *input, complex fract16 *temp, complex_fract16 *output, const complex_fract16 *twiddle_table, int twiddle_stride, int fft_size, const complex_fract16 *twiddle_table, int twiddle_stride, int fft_size, int block_exponent, int scale_method)
N Point Radix-2 Real Input FFT	void rfft_fr16 (const fract16 *input, complex_fract16 *temp, complex_fract16 *output, const complex_fract16 *twiddle_table, int twiddle_stride, int fft_size, int block_exponent, int scale_method)
N Point Radix-2 Inverse FFT	void ifft_fr16 (const complex_fract16 *input, complex_fract16 *temp, complex_fract16 *output, const complex_fract16 *twiddle_table, int twiddle_stride, int fft_size, int block_exponent, int scale_method)
N Point Radix-4 Complex Input FFT	void cfftrad4_fr16 (const complex_fract16 *input, complex fract16 *temp, complex_fract16 *output, const complex_fract16 *twiddle_table, int twiddle_stride, int fft_size, int block_exponent, int scale_method)
N Point Radix-4 Real Input FFT	void rfftrad4_fr16 (const fract16 *input, complex_fract16 *temp, complex_fract16 *output, const complex_fract16 *twiddle_table, int twiddle_stride, int fft_size, int block_exponent, int scale_method)

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

Description	Prototype

N Point Radix-4
Inverse Input FFT
	`void ifftrad4_fr16`
	`(const complex_fract *input,`
	`complex_fract16 *temp, complex_fract16 *output,`
	`const complex_fract16 *twiddle_table,`
	`int twiddle_stride, int fft_size,`
	`int block_exponent, int scale_method)`

Fast N point Radix-4
Complex Input FFT
	`void cfftf_fr16`
	`(const complex_fract16 *input,`
	`complex_fract16 *output,`
	`const complex_fract16 *twiddle_table,`
	`int twiddle_stride, int fft_size)`

Nxn Point 2-D
Complex Input FFT
	`void cfft2d_fr16`
	`(const complex_fract16 *input,`
	`complex fract16 *temp, complex_fract16 *output,`
	`const complex_fract16 *twiddle_table,`
	`int twiddle_stride, int fft_size,`
	`int block_exponent, int scale_method)`

Nxn Point 2-D
Real Input FFT
	`void rfft2d_fr16`
	`(const fract16 *input, complex_fract16 *temp,`
	`complex_fract16 *output,`
	`const complex_fract16 *twiddle_table,`
	`int twiddle_stride, int fft_size,`
	`int block_exponent, int scale_method)`

Nxn Point 2-D
Inverse FFT
	`void ifft2d_fr16`
	`(const complex_fract16 *input,`
	`complex_fract16 *temp, complex_fract16 *output,`
	`const complex_fract16 *twiddle_table,`
	`int twiddle_stride, int fft_size,`
	`int block_exponent, int scale_method)`

Convolutions	

Convolution	`void convolve_fr16`
	`(const fract16 input_x[], int length_x,`
	`const fract16 input_y[], int length_y,`
	`fract16 output[])`

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

Description	Prototype
2-D Convolution	`void conv2d_fr16 (const fract16 *input_x, int rows_x, int columns_x, const fract16 *input_y, int rows_y, int columns_y, fract16 *output)`
2-D Convolution 3x3 Matrix	`void conv2d3x3_fr16 (const fract16 *input_x, int rows_x, int columns_x, const fract16 input_y [3] [3], fract16 *output)`
Compression/Expansion	
A-law compression	`void a_compress (const short input[], short output[], int length)`
A-law expansion	`void a_expand (const short input[], short output[], int length)`
µ-law compression	`void mu_compress (const short input[], short output[], int length)`
µ-law expansion	`void mu_expand (const char input[], short output[], int length)`

math.h – Math Functions

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

Table 4-5 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 "C Run-Time Library Reference" on page 3-60.

This header file also provides prototypes for a number of additional math functions—`clip`, `copysign`, `max`, and `min`, and an integer function, `countones`. These functions are described in "DSP Run-Time Library Reference" on page 4-46.

Table 4-5. Math Library

Description	Prototype
Absolute Value	double fabs (double x) float fabsf (float x) long double fabsd (long double x)
Anti-log	double alog (double x) float alogf (float x) long double alogd (long double x)
Base 10 Anti-log	double alog10 (double x) float alog10f (float x) long double alog10d (long double x)
Arc Cosine	double acos (double x) float acosf (float x) long double acosd (long double x) fract16 acos_fr16 (fract16 x)
Arc Sine	double asin (double x) float asinf (float x) long double asind (long double x) fract16 asin_fr16 (fract16 x)
Arc Tangent	double atan (double x) float atanf (float x) long double atand (long double x) fract16 atan_fr16 (fract16 x)
Arc Tangent of Quotient	double atan2 (double x, double y) float atan2f (float x, float y) long double atan2d (long double x, long double y) fract16 atan2_fr16 (fract16 x, fract16 y)
Ceiling	double ceil (double x) float ceilf (float x) long double ceild (long double x)
Cosine	double cos (double x) float cosf (float x) long double cosd (long double x) fract16 cos_fr16 (fract16 x)

Table 4-5. Math Library (Cont'd)

Description	Prototype

Cotangent	`double cot (double x)`
	`float cotf (float x)`
	`long double cotd (long double x)`

Hyperbolic Cosine	`double cosh (double x)`
	`float coshf (float x)`
	`long double coshd (long double x)`

Exponential	`double exp (double x)`
	`float expf (float x)`
	`long double expd (long double x)`

Floor	`double floor (double x)`
	`float floorf (float x)`
	`long double floord (long double x)`

Floating-Point Remainder	`double fmod (double x, double y)`
	`float fmodf (float x, float y)`
	`long double fmodd (long double x, long double y)`

Get Mantissa and Exponent	`double frexp (double x, int *n)`
	`float frexpf (float x, int *n)`
	`long double frexpd (long double x, int *n)`

Is Not A Number?	`int isnanf (float x)`
	`int isnan (double x)`
	`int isnand (long double x)`

Is Infinity?	`int isinff (float x)`
	`int isinf (double x)`
	`int isinfd (long double x)`

Multiply by Power of 2	`double ldexp(double x, int n)`
	`float ldexpf(float x, int n)`
	`long double ldexpd (long double x, int *n)`

Natural Logarithm	`double log (double x)`
	`float logf (float x)`
	`long double logd (long double x)`

Logarithm Base 10	`double log10 (double x)`
	`float log10f (float x)`
	`long double log10d (long double x)`

Table 4-5. Math Library (Cont'd)

Description	Prototype

Get Fraction and Integer	`double modf (double x, double *i)`
	`float modff (float x, float *i)`
	`long double modfd (long double x, long double *i)`

Power	`double pow (double x, double y)`
	`float powf (float x, float y)`
	`long double powd (long double x, long double y)`

Reciprocal Square Root	`double rsqrt (double x)`
	`float rsqrtf (float x)`
	`long double rsqrtd (long double x)`

Sine	`double sin (double x)`
	`float sinf (float x)`
	`long double sind (long double x)`
	`fract16 sin_fr16 (fract16 x)`

Hyperbolic Sine	`double sinh (double x)`
	`float sinhf (float x)`
	`long double sinhd (long double x)`

Square Root	`double sqrt (double x)`
	`float sqrtf (float x)`
	`long double sqrtd (long double x)`
	`fract16 sqrt_fr16 (fract16 x)`

Tangent	`double tan (double x)`
	`float tanf (float x)`
	`long double tand (long double x)`
	`fract16 tan_fr16 (fract16 x)`

Hyperbolic Tangent	`double tanh (double x)`
	`float tanhf (float x)`
	`long double tanhd (long double x)`

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 4-5 for definitions of the complex types.

The matrix functions defined in the `matrix.h` header file are listed in Table 4-6. All the matrix functions that operate on the `complex_fract16` data type use saturating arithmetic.

Table 4-6. Matrix Functions

Description	Prototype

Real Matrix + Scalar Addition	`void matsadd (const double *matrix, double scalar, int rows, int columns, double *out)`
	`void matsaddf (const float *matrix, float scalar, int rows, int columns, float *out)`
	`void matsaddd (const long double *matrix, long double scalar, int rows, int columns, long double *out)`
	`void matsadd_fr16 (const fract16 *matrix, fract16 scalar, int rows, int columns, fract16 *out)`

Real Matrix – Scalar Subtraction	`void matssub (const double *matrix, double scalar, int rows, int columns, double *out)`
	`void matssubf (const float *matrix, float scalar, int rows, int columns, float *out)`
	`void matssubd (const long double *matrix, long double scalar, int rows, int columns, long double *out)`
	`void matssub_fr16 (const fract16 *matrix, fract16 scalar, int rows, int columns, fract16 *out)`

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

Description	Prototype

Real Matrix * Scalar Multiplication	`void matsmlt (const double *matrix, double scalar, int rows, int columns, double *out)`
	`void matsmltf (const float *matrix, float scalar, int rows, int columns, float *out)`
	`void matsmltd (const long double *matrix, long double scalar, int rows, int columns, long double *out)`
	`void matsmlt_fr16 (const fract16 *matrix, fract16 scalar, int rows, int columns, fract16 *out)`

Real Matrix + Matrix Addition	`void matmadd (const double *matrix_a, const double *matrix_b, int rows, int columns, double *out)`
	`void matmaddf (const float *matrix_a, const float *matrix_b, int rows, int columns, float *out)`
	`void matmaddd (const long double *matrix_a, const long double *matrix_b, int rows, int columns, long double *out)`
	`void matmadd_fr16 (const fract16 *matrix_a, const fract16 *matrix_b, int rows, int columns, fract16 *out)`

Real Matrix – Matrix Subtraction	`void matmsub (const double *matrix_a, const double *matrix_b, int rows, int columns, double *out)`
	`void matmsubf (const float *matrix_a, const float *matrix_b, int rows, int columns, float *out)`
	`void matmsubd (const long double *matrix_a, const long double *matrix_b, int rows, int columns, long double *out)`
	`void matmsub_fr16 (const fract16 *matrix_a, const fract16 *matrix_b, int rows, int columns, fract16 *out)`

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

Description	Prototype

Real Matrix * Matrix Multiplication	`void matmmlt (const double *matrix_a, int rows_a, int columns_a, const double *matrix_b, int columns_b, double *out)`
	`void matmmltf (const float *matrix_a, int rows_a, int columns_a, const float *matrix_b, int columns_b, float *out)`
	`void matmmltd (const long double *matrix_a, int rows_a, int columns_a, const long double *matrix_b, int columns_b, long double *out)`
	`void matmmlt_fr16 (const fract16 *matrix_a, int rows_a, int columns_a, const fract16 *matrix_b, int columns_b, fract16 *out)`

Complex Matrix + Scalar Addition	`void cmatsadd (const complex_double *matrix, complex_double scalar, int rows, int columns, complex_double *out)`
	`void cmatsaddf (const complex_float *matrix, complex_float scalar, int rows, int columns, complex_float *out)`
	`void cmatsaddd (const complex_long_double *matrix, complex_long_double scalar, int rows, int columns, complex_long_double *out)`
	`void cmatsadd_fr16 (const complex_fract16 *matrix, complex_fract16 scalar, int rows, int columns, complex_fract16 *out)`

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

Description	Prototype

Complex Matrix –
Scalar Subtraction
	`void cmatssub`
	`  (const complex_double *matrix,`
	`    complex_double scalar,`
	`    int rows, int columns, complex_double *out)`
	`void cmatssubf`
	`  (const complex_float *matrix,`
	`    complex_float scalar,`
	`    int rows, int columns, complex_float *out)`
	`void cmatssubd`
	`  (const complex_long_double *matrix,`
	`    complex_long_double scalar,`
	`    int rows, int columns, complex_long_double *out)`
	`void cmatssub_fr16`
	`  (const complex_fract16 *matrix,`
	`    complex_fract16 scalar,`
	`    int rows, int columns, complex_fract16 *out)`

Complex Matrix *
Scalar Multiplication
	`void cmatsmlt`
	`  (const complex_double *matrix,`
	`    complex_double scalar,`
	`    int rows, int columns, complex_double *out)`
	`void cmatsmltf`
	`  (const complex_float *matrix,`
	`    complex_float scalar,`
	`    int rows, int columns, complex_float *out)`
	`void cmatsmltd`
	`  (const complex_long_double *matrix,`
	`    complex_long_double scalar,`
	`    int rows, int columns, complex_long_double *out)`
	`void cmatsmlt_fr16`
	`  (const complex_fract16 *matrix,`
	`    complex_fract16 scalar,`
	`    int rows, int columns, complex_fract16 *out)`

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

Description	Prototype
Complex Matrix + Matrix Addition	void cmatmadd (const complex_double *matrix_a, const complex_double *matrix_b, int rows, int columns, complex_double *out) void cmatmaddf (const complex_float *matrix_a, const complex_float *matrix_b, int rows, int columns, complex_float *out) void cmatmaddd (const complex_long_double *matrix_a, const complex_long_double *matrix_b, int rows, int columns, complex_long_double *out) void cmatmadd_fr16 (const complex_fract16 *matrix_a, const complex_fract16 *matrix_b, int rows, int columns, complex_fract16 *out)
Complex Matrix – Matrix Subtraction	void cmatmsub (const complex_double *matrix_a, const complex_double *matrix_b, int rows, int columns, complex_double *out) void cmatmsubf (const complex_float *matrix_a, const complex_float *matrix_b, int rows, int columns, complex_float *out) void cmatmsubd (const complex_long_double *matrix_a, const complex_long_double *matrix_b, int rows, int columns, complex_long_double *out) void cmatmsub_fr16 (const complex_fract16 *matrix_a, const complex_fract16 *matrix_b, int rows, int columns, complex_fract16 *out)

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

Description	Prototype

Complex Matrix * Matrix Multiplication	`void cmatmmlt (const complex_double *matrix_a, int rows_a, int columns_a, const complex_double *matrix_b, int columns_b, complex_double *out)`
	`void cmatmmltf (const complex_float *matrix_a, int rows_a, int columns_a, const complex_float *matrix_b, int columns_b, complex_float *out)`
	`void cmatmmltd (const complex_long_double *matrix_a, int rows_a, int columns_a, const complex_long_double *matrix_b, int columns_b, complex_long_double *out)`
	`void cmatmmlt_fr16 (const complex_fract16 *matrix_a, int rows_a int columns_a, const complex_fract16 *matrix_b, int columns_b, complex_fract16 *out)`

Transpose	`void transpm (const double *matrix, int rows, int columns, double *out)`
	`void transpmf (const float *matrix, int rows, int columns, float *out)`
	`void transpmd (const long double *matrix, int rows, int columns, long double *out)`
	`void transpm_fr16 (const fract16 *matrix, int rows, int columns, fract16 *out)`

In most of the function prototypes:

`*matrix_a` is a pointer to input matrix `matrix_a [] []`

`*matrix_b` is a pointer to input matrix `matrix_b [] []`

`scalar` is an input scalar

`rows` is the number of rows

`columns` is the number of columns

`*out` is a pointer to output matrix `out [] []`

In the `matrix*matrix` functions, `rows_a` and `columns_a` are the dimensions of matrix `a` and `rows_b` and `columns_b` are the dimensions of matrix `b`.

The functions described by this header assume that input array arguments are constant; that is, 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.

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 “DSP Run-Time Library Reference” on page 4-46.

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 4-5 for definitions of the complex types.

The functions defined in the `vector.h` header file are listed in Table 4-8. All the vector functions that operate on the `complex_fract16` data type use saturating arithmetic.

In the **Prototype** column, `vec[]`, vec_a[] and `vec_b[]` are input vectors, `scalar` is an input scalar, `out[]` is an output vector, and `sample_length` is the number of elements.The functions assume that input array arguments are constant; that is, their contents will not change during the course of

Table 4-7. Statistical Functions

Description	Prototype
Autocoherence	void autocohf (const float samples[], int sample_length, int lags, float out[]) void autocoh (const double samples[], int sample_length, int lags, double out[]) void autocohd (const long double samples[], int sample_length, int lags, long double out[]) void autocoh_fr16 (const fract16 samples[], int sample_length, int lags, fract16 out[])
Autocorrelation	void autocorrf (const float samples[], int sample_length, int lags, float out[]) void autocorr (const double samples[], int sample_length, int lags, double out[]) void autocorrd (const long double a[], int sample_length, int lags, long double out[]) void autocorr_fr16 (const fract16 samples[], int sample_length, int lags, fract16 out[])
Cross-coherence	void crosscohf (const float samples_a[], const float samples_b[], int sample_length, int lags, float out[]) void crosscoh (const double samples_a[], const double samples_b[], int sample_length, int lags, double out[]) void crosscohd (const long double samples_a[], const long double samples_b[], int sample_length int lags, long double out[]) void crosscoh_fr16 (const fract16 samples_a[], const fract16 samples_b[], int sample_length, int lags, fract16 out[])

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

Description	Prototype

Cross-correlation	`void crosscorrf (const float samples_a[], const float samples_b[], int sample_length, int lags, float out[])`
	`void crosscorr (const double samples_a[], const double samples_b[], int sample_length, int lags, double out[])`
	`void crosscorrd (const long double samples_a[], const long double samples_b[], int sample_length, int lags, long double out[])`
	`void crosscorr_fr16 (const fract16 samples_a[], const fract16 samples_b[], int sample_length, int lags, fract16 out[])`

Histogram	`void histogramf (const float samples_a[], int out[], float max_sample, float min_sample, int sample_length, int bin_count)`
	`void histogram (const double samples_a[], int out[], double max_sample, double min_sample, int sample_length, int bin_count)`
	`void histogramd (const long double samples_a[], int out[], long double max_sample, long double min_sample, int sample_length, int bin_count)`
	`void histogram_fr16 (const fract16 samples_a[], int out[], fract16 max_sample, fract16 min_sample, int sample_length, int bin_count)`

Mean	`float meanf (const float samples[], int sample_length)`
	`double mean (const double samples[], int sample_length)`
	`long double meand (const long double samples[], int sample_length)`
	`fract16 mean_fr16 (const fract16 samples[], int sample_length)`

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

Description	Prototype

Root Mean Square	`float rmsf (const float samples[], int sample_length)`
	`double rms (const double samples[], int sample_length)`
	`long double rmsd`
	`(const long double samples[], int sample_length)`
	`fract16 rms_fr16`
	`(const fract16 samples[], int sample_length)`

Variance	`float varf (const float samples[], int sample_length)`
	`double var (const double samples[], int sample_length)`
	`long double vard`
	`(const long double samples[], int sample_length)`
	`fract16 var_fr16`
	`(const fract16 samples[], int sample_length)`

Count Zero Crossing	`int zero_crossf`
	`(const float samples[], int sample_length)`
	`int zero_cross`
	`(const double samples[], int sample_length)`
	`int zero_crossd`
	`(const long double samples[], int sample_length)`
	`int zero_cross_fr16`
	`(const fract16 samples[], int sample_length)`

the routine. In particular, this means the input arguments do not overlap with any output argument. In general, better run-time performance is achieved by the vector functions if the input vectors and the output vector are in different memory banks. This structure avoids any potential memory bank collisions.

window.h – Window Generators

The `window.h` header file contains various functions to generate windows based on various methodologies. The functions defined in the `window.h` header file are listed in Table 4-9 and are described in detail in "DSP Run-Time Library Reference" on page 4-46.

Table 4-8. Vector Functions

Description	Prototype

Real Vector +
Scalar Addition
	`void vecsadd`
	`(const double vec[], double scalar,`
	`double out[], int length)`
	`void vecsaddd`
	`(const long double vec[], long double scalar,`
	`long double out[], int length)`
	`void vecsaddf`
	`(const float vec[], float scalar,`
	`float out[], int length)`
	`void vecsadd_fr16`
	`(const fract16 vec[], fract16 scalar,`
	`fract16 out[], int length)`

Real Vector –
Scalar Subtraction
	`void vecssub`
	`(const double vec[], double scalar,`
	`double out[], int length)`
	`void vecssubd`
	`(const long double vec[], long double scalar,`
	`long double out[], int length)`
	`void vecssubf`
	`(const float vec[], float scalar,`
	`float out[], int length)`
	`void vecssub_fr16`
	`(const fract16 vec[], fract16 scalar,`
	`fract16 out[], int length)`

Real Vector *
Scalar Multiplication
	`void vecsmlt`
	`(const double vec[], double scalar,`
	`double out[], int length)`
	`void vecsmltd`
	`(const long double vec[], long double scalar,`
	`long double out[], int length)`
	`void vecsmltf`
	`(const float vec[], float scalar,`
	`float out[], int length)`
	`void vecsmlt_fr16`
	`(const fract16 vec[], fract16 scalar,`
	`fract16 out[], int length)`

Table 4-8. Vector Functions (Cont'd)

Description	Prototype

Real Vector +
Vector Addition
	`void vecvadd`
	`  (const double vec_a[], const double vec_b[],`
	`    double out[], int length)`
	`void vecvaddd`
	`  (const long double vec_a[],`
	`    const long double vec_b[],`
	`    long double out[], int length)`
	`void vecvaddf`
	`  (const float vec_a[], const float vec_b[],`
	`    float out[], int length)`
	`void vecvadd_fr16`
	`  (const fract16 vec_a[], const fract16 vec_b[],`
	`    fract16 out[], int length)`

Real Vector –
Vector Subtraction
	`void vecvsub`
	`  (const double vec_a[], const double vec_b[ ],`
	`    double out[], int length)`
	`void vecvsubd`
	`  (const long double vec_a[], const long double`
	`vec_b[],`
	`    long double out[], int length)`
	`void vecvsubf`
	`  (const float vec_a[], const float vec_b[],`
	`    float out[], int length)`
	`void vecvsub_fr16`
	`  (const fract16 vec_a[],`
	`    const fract16 vec_b[],`
	`    fract16 vec_b[], fract16 out[], int length)`

Table 4-8. Vector Functions (Cont'd)

Description	Prototype

Real Vector * Vector Multiplication	`void vecvmlt`
	`(const double vec_a[], const double vec_b[],`
	`double out[], int length)`
	`void vecvmltd`
	`(const long double vec_a[], const long double`
	`vec_b[],`
	`long double out[], int length)`
	`void vecvmltf`
	`(const float vec_a[], const float vec_b[],`
	`float out[], int length)`
	`void vecvmlt_fr16`
	`(const fract16 vec_a[], const fract16 vec_b[],`
	`fract16 out[], int length)`

Maximum Value of Vector Elements	`double vecmax (const double vec[], int length)`
	`long double vecmaxd`
	`(const long double vec[], int length)`
	`float vecmaxf (const float vec[], int length)`
	`fract16 vecmax_fr16`
	`(const fract16 vec[], int length)`

Minimum Value of Vector Elements	`double vecmin (const double vec[], int length)`
	`long double vecmind`
	`(const long double vec[], int length)`
	`float vecminf (const float vec[], int length)`
	`fract16 vecmin_fr16(const fract16 vec[], int length)`
	`fract16 vecmin_fr16(const fract16 vec[], int length)`

Index of Maximum Value of Vector Elements	`int vecmaxloc (const double vec[], int length)`
	`int vecmaxlocd`
	`(const long double vec[], int length)`
	`int vecmaxlocf(const float vec[], int length);`
	`int vecmaxloc_fr16`
	`(const fract16 vec[], int length)`

Index of Minimum Value of Vector Elements	`int vecminloc (const double vec[], int length)`
	`int vecminlocd(const long double vec[], int length)`
	`int vecminlocf (const float vec[], int length)`
	`int vecminloc_fr16(const fract16 vec[], int length)`

Table 4-8. Vector Functions (Cont'd)

Description	Prototype

Complex Vector + Scalar Addition	`void cvecsadd (const complex_double vec[], complex_double scalar, complex_double out[], int length)`
	`void cvecsaddd (const complex_long_double vec[], complex_long_double scalar, complex_long_double out[], int length)`
	`void cvecsaddf (const complex_float vec[], complex_float scalar, complex_float out[], int length)`
	`void cvecsadd_fr16 (const complex_fract16 vec[], complex_fract16 scalar, complex_fract16 out[], int length)`

Complex Vector – Scalar Subtraction	`void cvecssub (const complex_double vec[], complex_double scalar, complex_double out[], int length)`
	`void cvecssubd (const complex_long_double vec[], complex_long_double scalar, complex_long_double out[], int length)`
	`void cvecssubf (const complex_float vec[], complex_float scalar, complex_float out[], int length)`
	`void cvecssub_fr16 (const complex_fract16 vec[], complex_fract16 scalar, complex_fract16 out[], int length)`

Table 4-8. Vector Functions (Cont'd)

Description	Prototype

Complex Vector * Scalar Multiplication	`void cvecsmlt(`
	`  (const complex_double vec[],`
	`    complex_double scalar,`
	`    complex_double out[], int length)`
	`void cvecsmltd(`
	`  (const complex_long_double vec[],`
	`    complex_long_double scalar,`
	`    complex_long_double out[], int length)`
	`void cvecsmltf`
	`  (const complex_float vec[],`
	`    complex_float scalar,`
	`    complex_float out[], int length)`
	`void cvecsmlt_fr16`
	`  (const complex_fract16 vec[],`
	`    complex_fract16 scalar,`
	`    complex_fract16 out[], int length)`

Complex Vector + Vector Addition	`void cvecvadd`
	`  (const complex_double vec_a[],`
	`    const complex_double vec_b[],`
	`    complex_double out[], int length)`
	`void cvecvaddd`
	`  (const complex_long_double vec_a[],`
	`    const complex_long_double vec_b[],`
	`    complex_long_double out[], int length)`
	`void cvecvaddf`
	`  (const complex_float vec_a[],`
	`    const complex_float vec_b[],`
	`    complex_float out[], int length)`
	`void cvecvadd_fr16`
	`  (const complex_fract16 vec_a[],`
	`    const complex_fract16 vec_b[],`
	`    complex_fract16 out[], int length)`

Table 4-8. Vector Functions (Cont'd)

Description	Prototype

Complex Vector – Vector Subtraction	`void cvecvsub (const complex_double vec_a[], const complex_double vec_b[], complex_double out[], int length)`
	`void cvecvsubd (const complex_long_double vec_a[], const complex_long_double vec_b[], complex_long_double out[], int length)`
	`void cvecvsubf (const complex_float vec_a[], const complex_float vec_b[], complex_float out[], int length)`
	`void cvecvsub_fr16 (const complex_fract16 vec_a[], const complex_fract16 vec_b[], complex_fract16 out[], int length)`

Complex Vector * Vector Multiplication	`void cvecvmlt (const complex_double vec_a[], const complex_double vec_b[], complex_double out[], int length)`
	`void cvecvmltd (const complex_long_double vec_a[], const complex_long_double vec_b[], complex_long_double out[], int length)`
	`void cvecvmltf (const complex_float vec_a[], const complex_float vec_b[], complex_float out[], int length)`
	`void cvecvmlt_fr16 (const complex_fract16 vec_a[], const complex_fract16 vec_b[], complex_fract16 out[], int length)`

Table 4-8. Vector Functions (Cont'd)

Description	Prototype

Real Vector Dot Product	`double vecdot (const double vec_a[], const double vec_b[], int length)`
	`long double vecdotd (const long double vec_a[], const long double vec_b[], int length)`
	`float vecdotf (const float vec_a[], const float vec_b[], int length)`
	`fract16 vecdot_fr16 (const fract16 vec_a[], const fract16 vec_b[], int length)`

Complex Vector Dot Product	`complex_double cvecdot (const complex_double vec_a[], const complex_double vec_b[], int length)`
	`complex_long_double cvecdotd (const complex_long_double vec_a[], const complex_long_double vec_b[], int length)`
	`complex_float cvecdotf (const complex_float vec_a[], const complex_float vec_b[], int length)`
	`complex_fract16 cvecdot_fr16 (const complex_fract16 vec_a[], const complex_fract16 vec_b[], int length)`

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

Table 4-9. Window Generator Functions

Description	Prototype
Generate Bartlett Window	`void gen_bartlett_fr16 (fract16 bartlett_window[], int window_stride, int window_size)`
Generate Blackman Window	`void gen_blackman_fr16 (fract16 blackman_window[], int window_stride, int window_size)`
Generate Gaussian Window	`void gen_gaussian_fr16 (fract16 gaussian_window[], float alpha, int window_stride, int window_size)`
Generate Hamming Window	`void gen_hamming_fr16 (fract16 hamming_window[], int window_stride, int window_size)`
Generate Hanning Window	`void gen_hanning_fr16 (fract16 hanning_window[], int window_stride, int window_size)`
Generate Harris Window	`void gen_harris_fr16 (fract16 harris_window[], int window_stride, int window_size)`
Generate Kaiser Window	`void gen_kaiser_fr16 (fract16 kaiser_window[], int window_stride, int window_size)`
Generate Rectangular Window	`void gen_rectangular_fr16 (fract16 rectangular_window[], int window_stride, int window_size)`
Generate Triangle Window	`void gen_triangle_fr16 (fract16 triangle_window[], int window_stride, int window_size)`
Generate Vonhann Window	`void gen_vonhann_fr16 (fract16 vonhann_window[], int window_stride, int window_size)`

Measuring Cycle Counts

The common basis for benchmarking some arbitrary C-written source is to measure the number of processor cycles that the code uses. Once this figure is known, it can be used to calculate the actual time taken by multiplying the number of processor cycles by the clock rate of the processor. The run-time library provides three alternative methods for measuring processor counts. Each of these methods is described in the following sections:

- "Basic Cycle Counting Facility" on page 4-36
- "Cycle Counting Facility with Statistics" on page 4-38
- "Using time.h to Measure Cycle Counts" on page 4-41
- "Determining the Processor Clock Rate" on page 4-43
- "Considerations when Measuring Cycle Counts" on page 4-44

Basic Cycle Counting Facility

The fundamental approach to measuring the performance of a section of code is to record the current value of the cycle count register before executing the section of code, and then reading the register again after the code has been executed. This process is represented by two macros that are defined in the `cycle_count.h` header file; the macros are:

```
START_CYCLE_COUNT(S)

STOP_CYCLE_COUNT(T,S)
```

The parameter `S` is set by the macro `START_CYCLE_COUNT` to the current value of the cycle count register; this value should then be passed to the macro `STOP_CYCLE_COUNT`, which will calculate the difference between the parameter and current value of the cycle count register. Reading the cycle count register incurs an overhead of a small number of cycles and the macro ensures that the difference returned (in the parameter T) will be adjusted to allow for this additional cost. The parameters S and T must be separate variables; they should be declared as a cycle_t data type which the header file cycle_count.h defines as:

```
typedef volatile unsigned long long cycle_t;
```

The header file also defines the macro:

```
PRINT_CYCLES(STRING,T)
```

which is provided mainly as an example of how to print a value of type cycle_t; the macro outputs the text STRING on stdout followed by the number of cycles T.

The instrumentation represented by the macros defined in this section is only activated if the program is compiled with the -DDO_CYCLE_COUNTS switch. If this switch is not specified, then the macros are replaced by empty statements and have no effect on the program.

The following example demonstrates how the basic cycle counting facility may be used to monitor the performance of a section of code:

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

extern int
main(void)
{
   cycle_t start_count;
   cycle_t final_count;

   START_CYCLE_COUNT(start_count)
   Some_Function_Or_Code_To_Measure();
   STOP_CYCLE_COUNT(final_count,start_count)
```

```
   PRINT_CYCLES("Number of cycles: ",final_count)
}
```

The run-time libraries provide alternative facilities for measuring the performance of C source (see "Cycle Counting Facility with Statistics" on page 4-38 and "Using time.h to Measure Cycle Counts" on page 4-41); the relative benefits of this facility are outlined in "Considerations when Measuring Cycle Counts" on page 4-44.

The basic cycle counting facility is based upon macros; it may therefore be customized for a particular application if required, without the need for rebuilding the run-time libraries.

Cycle Counting Facility with Statistics

The `cycles.h` header file defines a set of macros for measuring the performance of compiled C source. As well as providing the basic facility for reading the cycle count registers of the Blackfin architecture, the macros also have the capability of accumulating statistics that are suited to recording the performance of a section of code that is executed repeatedly.

If the switch `-DDO_CYCLE_COUNTS` is specified at compile-time, then the `cycles.h` header file defines the following macros:

- `CYCLES_INIT(S)`
  a macro that initializes the system timing mechanism and clears the parameter `S`; an application must contain one reference to this macro.

- `CYCLES_START(S)`
  a macro that extracts the current value of the cycle count register and saves it in the parameter `S`.

- `CYCLES_STOP(S)`
  a macro that extracts the current value of the cycle count register and accumulates statistics in the parameter `S`, based on the previous reference to the `CYCLES_START` macro.

- `CYCLES_PRINT(S)`
  a macro which prints a summary of the accumulated statistics recorded in the parameter `S`.

- `CYCLES_RESET(S)`
  a macro which re-zeros the accumulated statistics that are recorded in the parameter `S`.

The parameter `S` that is passed to the macros must be declared to be of the type `cycle_stats_t`; this is a structured data type that is defined in the `cycles.h` header file. The data type has the capability of recording the number of times that an instrumented part of the source has been executed, as well as the minimum, maximum, and average number of cycles that have been used. If an instrumented piece of code has been executed for example, 4 times, the `CYCLES_PRINT` macro would generate output on the standard stream `stdout` in the form:

```
AVG   : 95

MIN   : 92

MAX   : 100

CALLS : 4
```

If an instrumented piece of code had only been executed once, then the `CYCLES_PRINT` macro would print a message of the form:

```
CYCLES : 95
```

If the switch `-DDO_CYCLE_COUNTS` is not specified, then the macros described above are defined as null macros and no cycle count information is gathered. To switch between development and release mode therefore only requires a re-compilation and will not require any changes to the source of an application.

The macros defined in the `cycles.h` header file may be customized for a particular application without the requirement for rebuilding the run-time libraries.

An example that demonstrates how this facility may be used is:

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

extern void foo(void);
extern void bar(void);

extern int
main(void)
{
   cycle_stats_t stats;
   int i;

   CYCLES_INIT(stats)

   for (i = 0; i < LIMIT; i++) {
      CYCLES_START(stats)
      foo();
      CYCLES_STOP(stats)
   }
   printf("Cycles used by foo\n");
   CYCLES_PRINT(stats)
   CYCLES_RESET(stats)

   for (i = 0; i < LIMIT; i++) {
      CYCLES_START(stats)
      bar();
      CYCLES_STOP(stats)
   }
   printf("Cycles used by bar\n");
```

```
    CYCLES_PRINT(stats)
}
```

This example might output:

```
    Cycles used by foo

        AVG   : 25454

        MIN   : 23003

        MAX   : 26295

        CALLS : 16

    Cycles used by bar

        AVG   : 8727

        MIN   : 7653

        MAX   : 8912

        CALLS : 16
```

Alterative methods of measuring the performance of compiled C source are described in the sections "Basic Cycle Counting Facility" on page 4-36 and "Using time.h to Measure Cycle Counts" on page 4-41. Also refer to "Considerations when Measuring Cycle Counts" on page 4-44 which provides some useful tips with regards to performance measurements.

Using time.h to Measure Cycle Counts

The `time.h` header file defines the data type `clock_t`, the `clock` function, and the macro `CLOCKS_PER_SEC`, which together may be used to calculate the number of seconds spent in a program.

In the ANSI C standard, the `clock` function is defined to return the number of implementation dependent clock "ticks" that have elapsed since the program began, and in this version of the C/C++ compiler the function returns the number of processor cycles that an application has used.

The conventional way of using the facilities of the `time.h` header file to measure the time spent in a program is to call the `clock` function at the start of a program, and then subtract this value from the value returned by a subsequent call to the function. This difference is usually cast to a floating-point type, and is then divided by the macro `CLOCKS_PER_SEC` to determine the time in seconds that has occurred between the two calls.

If this method of timing is used by an application then it is important to note that:

- the value assigned to the macro `CLOCKS_PER_SEC` should be independently verified to ensure that it is correct for the particular processor being used (see "Determining the Processor Clock Rate" on page 4-43),
- the result returned by the `clock` function does not include the overhead of calling the library function.

A typical example that demonstrates the use of the `time.h` header file to measure the amount of time that an application takes is shown below.

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

extern int
main(void)
{
    volatile clock_t clock_start;
    volatile clock_t clock_stop;

    double secs;
```

```
    clock_start = clock();
    Some_Function_Or_Code_To_Measure();
    clock_stop = clock();

    secs = ((double) (stop_time - start_time))
           / CLOCKS_PER_SEC;
    printf("Time taken is %e seconds\n",secs);
}
```

The header files `cycles.h` and `cycle_count.h` define other methods for benchmarking an application—these header files are described in the sections "Basic Cycle Counting Facility" on page 4-36 and "Cycle Counting Facility with Statistics" on page 4-38, respectively. Also refer to "Considerations when Measuring Cycle Counts" on page 4-44 which provides some guidelines that may be useful.

Determining the Processor Clock Rate

Applications may be benchmarked with respect to how many processor cycles that they use. However, more typically applications are benchmarked with respect to how much time (for example, in seconds) that they take.

To measure the amount of time that an application takes to run on a Blackfin processor usually involves first determining the number of cycles that the processor takes, and then dividing this value by the processor's clock rate. The `time.h` header file defines the macro `CLOCKS_PER_SEC` as the number of processor "ticks" per second. On Blackfin processors, it is set by the run-time library to one of the following values in descending order of precedence:

- via the compile-time switch `-DCLOCKS_PER_SEC=<definition>`. Because the `time_t` type is based on the `long long int` data type, it is recommended that the value assigned to the symbolic name

CLOCKS_PER_SEC is defined as the same data type by qualifying the value with the LL (or ll) suffix (for example, -DCLOCKS_PER_SEC=600000000LL).

- via the System Services Library
- via the Processor speed box in the VisualDSP++ Project Options dialog box, Compile tab, Processor (1) category
- from the **cycles.h** header file

If the value of the macro CLOCKS_PER_SEC is taken from the cycles.h header file, then be aware that the clock rate of the processor will usually be taken to be the maximum speed of the processor, which is not necessarily the speed of the processor at RESET.

Considerations when Measuring Cycle Counts

The run-time library provides three different methods for benchmarking C-compiled code. Each of these alternatives are described in the sections:

- "Basic Cycle Counting Facility" on page 4-36
  The basic cycle counting facility represents an inexpensive and relatively inobtrusive method for benchmarking C-written source using cycle counts. The facility is based on macros that factor-in the overhead incurred by the instrumentation. The macros may be customized and they can be switched either or off, and so no source changes are required when moving between development and release mode. The same set of macros is available on other platforms provided by Analog Devices.

- "Cycle Counting Facility with Statistics" on page 4-38
  This is a cycle-counting facility that has more features than the basic cycle counting facility described above. It is therefore more expensive in terms of program memory, data memory, and cycles consumed. However, it does have the ability to record the number of times that the instrumented code has been executed and to cal-

culate the maximum, minimum, and average cost of each iteration. The macros provided take into account the overhead involved in reading the cycle count register. By default, the macros are switched off, but they are switched on by specifying the `-DDO_CYCLE_COUNTS` compile-time switch. The macros may also be customized for a specific application. This cycle counting facility is also available on other Analog Devices architectures.

- "Using time.h to Measure Cycle Counts" on page 4-41
  The facilities of the `time.h` header file represent a simple method for measuring the performance of an application that is portable across a large number of different architectures and systems. These facilities are based around the `clock` function.

  The `clock` function however does not take into account the cost involved in invoking the function. In addition, references to the function may affect the code that the optimizer generates in the vicinity of the function call. This method of benchmarking may not accurately reflect the true cost of the code being measured.

  This method is more suited to benchmarking applications rather than smaller sections of code that run for a much shorter time span.

  When benchmarking code, some thought is required when adding timing instrumentation to C source that will be optimized. If the sequence of statements to be measured is not selected carefully, the optimizer may move instructions into (and out of) the code region and/or it may re-site the instrumentation itself, thus leading to distorted measurements. It is therefore generally considered more reliable to measure the cycle count of calling (and returning from) a function rather than a sequence of statements within a function.

  It is recommended that variables that are used directly in benchmarking are simple scalars that are allocated in internal memory (be they assigned the result of a reference to the `clock` function, or be they used as arguments to the cycle counting macros). In the case of variables that are assigned the result of the `clock` function, it is also recommended that they be defined with the `volatile` keyword.

The cycle count registers of the Blackfin architecture are called the `CYCLES` and `CYCLES2` registers. These registers are 32-bit registers. The `CYCLES` register is incremented at every processor cycle; when it wraps back to zero the `CYCLES2` register is incremented. Together these registers represent a 64-bit counter that is unlikely to wrap around to zero during the timing of an application.

DSP Run-Time Library Reference

This section provides descriptions of the DSP run-time library functions.

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 – Required header file and functional prototype; when the functionality is provided for several data types (for example, `float`, `double`, `long double` or `fract16`), several prototypes are given

Description – Function specification

Algorithm – High-level mathematical representation of the function

Domain – Range of values supported by the function

Notes – Miscellaneous information

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

a_compress

A-law compression

Synopsis

```
#include <filter.h>
void a_compress(const short input[], short output[], int length);
```

Description

The `a_compress` function takes a vector of linear 13-bit signed speech samples and performs A-law compression according to ITU recommendation G.711. Each sample is compressed to 8 bits and is returned in the vector pointed to by `output`.

Algorithm

`C(k)=a-law` compression of $A(k)$ for k = 0 to `length`-1

Domain

Content of input array: –4096 to 4095

a_expand

A-law expansion

Synopsis

```
#include <filter.h>
void a_expand (const short input[], short output[], int length);
```

Description

The `a_expand` function inputs a vector of 8-bit compressed speech samples and expands them according to ITU recommendation G.711. Each input value is expanded to a linear 13-bit signed sample in accordance with the A-law definition and is returned in the vector pointed to by `output`.

Algorithm

`C(k)=a-law` expansion of A(*k*) for k = 0 to `length`-1

Domain

Content of input array: 0 to 255

alog

anti-log

Synopsis

```
#include <math.h>

float alogf (float x);
double alog (double x);
long double alogd (long double x);
```

Description

The `alog` functions calculate the natural (base `e`) anti-log of their argument. An anti-log function performs the reverse of a `log` function and is therefore equivalent to exponentiation.

The value `HUGE_VAL` is returned if the argument `x` is greater than the function's domain. For input values less than the domain, the functions return 0.0.

Algorithm

$c = e^x$

Domain

`x` = [–87.33 , 88.72] for `alogf()`

`x` = [–708.39 , 709.78] for `alogd()`

Example

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

```
double y;
y = alog(1.0);          /* y = 2.71828... */
```

See Also

alog10, exp, log, pow

alog10

base 10 anti-log

Synopsis

```
#include <math.h>

float alog10f (float x);
double alog10 (double x);
long double alog10d (long double x);
```

Description

The `alog10` functions calculate the base 10 anti-log of their argument. An anti-log function performs the reverse of a `log` function and is therefore equivalent to exponentiation. Therefore, `alog10(x)` is equivalent to `exp(x * log(10.0))`.

The value `HUGE_VAL` is returned if the argument `x` is greater than the function's domain. For input values less than the domain, the functions return 0.0.

Algorithm

$c = e^{(x * \log(10.0))}$

Domain

x = [–37.92 , 38.53] for `alog10f()`

x = [–307.65 , 308.25] for `alog10d()`

Example

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

```
double y;
y = alog10(1.0);         /* y = 10.0 */
```

See Also

alog, exp, log10, pow

arg

get phase of a complex number

Synopsis

```
#include <complex.h>

float argf (complex_float a);
double arg (complex_double a);
long double argd (complex_long_double a);
fract16 arg_fr16 (complex_fract16 a);
```

Description

These functions compute the phase associated with a Cartesian number, represented by the complex argument `a`, and return the result.

Refer to the description of the `polar_fr16` function which explains how a phase, represented as a fractional number, is interpreted in polar notation (see "polar" on page 4-139).

Algorithm

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

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$	for `argf( )`
-1.7×10^{308} to $+1.7 \times 10^{308}$	for `argd( )`
-1.0 to $+1.0$	for `arg_fr16( )`

Note

Im (a) /Re (a) < =1 for `arg_fr16 ( )`

autocoh

autocoherence

Synopsis

```
#include <stats.h>

void autocohf (const float  samples[ ],
               int          sample_length,
               int          lags,
               float        coherence[ ]);

void autocoh (const double  samples[ ],
              int           sample_length,
              int           lags,
              double        coherence[ ]);

void autocohd (const long double  samples[ ],
               int                sample_length,
               int                lags,
               long double        coherence[ ]);

void autocoh_fr16 (const fract16  samples[ ],
                   int            sample_length,
                   int            lags,
                   fract16        coherence[ ]);
```

Description

The autocoh functions compute the autocoherence of the input vector `samples[]`, which contain `sample_length` values. The autocoherence of an input signal is its autocorrelation minus its mean squared. The functions return the result in the output array `coherence[]` of length `lags`.

Algorithm

$$c_k = \frac{1}{n} * \sum_{j=0}^{n-k-1} (a_j * a_{j+k}) - (\overline{a})^2$$

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

Domain

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

-1.7×10^{308} to $+1.7 \times 10^{308}$ for `autocohd( )`

-1.0 to 1.0 for `autocoh_fr16( )`

autocorr

autocorrelation

Synopsis

```
#include <stats.h>

void autocorrf (const float  samples[ ],
                int          sample_length,
                int          lags,
                float        correlation[ ]);

void autocorr (const double  samples[ ],
                int          sample_length,
                int          lags,
                double       correlation[ ]);

void autocorrd (const long double  samples[ ],
                int                sample_length,
                int                lags,
                long double        correlation[ ]);

void autocorr_fr16 (const fract16  samples[ ],
                int                sample_length,
                int                lags,
                fract16            correlation[ ]);
```

Description

The autocorr functions perform an autocorrelation of a signal. Autocorrelation is the cross-correlation of a signal with a copy of itself. It provides information about the time variation of the signal. The signal to be autocorrelated is given by the `samples[]` input array. The number of samples of the autocorrelation sequence to be produced is given by `lags`. The length of the input sequence is given by `sample_length`.

Autocorrelation is used in digital signal processing applications such as speech analysis.

Algorithm

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

where a=`samples`; k = {0,1,...,m-1}; `m` is the number of `lags`; `n` is the size of the input vector `samples`.

Domain

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

-1.7×10^{308} to $+1.7 \times 10^{308}$ for `autocorrd( )`

-1.0 to $+ 1.0$ for `autocorr_fr16( )`

cabs

complex absolute value

Synopsis

```
#include <complex.h>

float cabsf (complex_float a);
double cabs (complex_double a);
long double cabsd (complex_long_double a);
fract16 cabs_fr16 (fract16 a);
```

Description

The cabs functions compute the complex absolute value of a complex input and return the result.

Algorithm

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

Domain

$\mathrm{Re}^2(a) + \mathrm{Im}^2(a) <= 3.4 \times 10^{38}$ for `cabsf( )`

$\mathrm{Re}^2(a) + \mathrm{Im}^2(a) <= 1.7 \times 10^{308}$ for `cabsd( )`

$\mathrm{Re}^2(a) + \mathrm{Im}^2(a) <= 1.0$ 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_long_double caddd (complex_long_double a,
                           complex_long_double b);
complex_fract16 cadd_fr16 (complex_fract16 a, complex_fract16 b);
```

Description

The cadd functions compute the complex addition of two complex inputs, `a` and `b`, and return the result.

Algorithm

```
Re(c) = Re(a) + Re(b)
Im(c) = Im(a) + Im(b)
```

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$	for `caddf( )`
-1.7×10^{308} to $+1.7 \times 10^{308}$	for `caddd( )`
-1.0 to $+1.0$	for `cadd_fr16( )`

cartesian

convert Cartesian to polar notation

Synopsis

```
#include <complex.h>

float cartesianf (complex_float  a, float  *phase);
double cartesian (complex_double a, double *phase);
long double  cartesiand (complex_long_double  a,
                         long double        *phase);
fract16 cartesian_fr16 (complex_fract16 a, fract16 *phase);
```

Description

The cartesian functions transform a complex number from Cartesian notation to polar notation. The Cartesian number is represented by the argument `a` that the function converts into a corresponding magnitude, which it returns as the function's result, and a phase that is returned via the second argument `phase`.

Refer to the description of the `polar_fr16` function which explains how a phase, represented as a fractional number, is interpreted in polar notation (see "polar" on page 4-139).

Algorithm

```
magnitude = cabs(a)

phase = arg(a)
```

Domain

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

-1.7×10^{308} to $+1.7 \times 10^{308}$ for `cartesiand( )`

-1.0 to +1.0 for `cartesian_fr16( )`

Example

```
#include <complex.h>

complex_float point = {-2.0 , 0.0};
float phase;
float mag;
mag = cartesianf (point,&phase);   /* mag = 2.0, phase = π */
```

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_long_double cdivd (complex_long_double a,
                           complex_long_double b);
complex_fract16 cdiv_fr16 (complex_fract16 a, complex_fract16 b);
```

Description

The cdiv functions compute the complex division of complex input `a` by complex input `b`, and return the result.

Algorithm

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

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

Domain

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

-1.7×10^{308} to $+1.7 \times 10^{308}$ for `cdivd( )`

-1.0 to 1.0 for `cdiv_fr16( )`

cexp

complex exponential

Synopsis

```
#include <complex.h>
complex_float cexpf (float x);
complex_double cexp (double x);
complex_long_double cexp (long double x);
```

Description

The cexp functions compute the complex exponential of real input `x` and return the result.

Algorithm

```
Re(c) = cos(x)
Im(c) = sin(x)
```

Domain

x = [–102940 ... 102940]   for `cexpf ( )`

x = [-8.433e8 ... 8.433e8]   for `cexpd ( )`

cfft

n point radix-2 complex input FFT

Synopsis

```
#include <filter.h>
void cfft_fr16(const complex_fract16 input[],
               complex_fract16       temp[],
               complex_fract16       output[],
               const complex_fract16 twiddle_table[],
               int                   twiddle_stride,
               int                   fft_size,
               int                   block_exponent,
               int                   scale_method);
```

Description

This function transforms the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT).

The size of the input array `input`, the output array `output`, and the temporary working buffer `temp` is `fft_size`, where `fft_size` represents the number of points in the FFT. By allocating these arrays in different memory banks, any potential data bank collisions are avoided, thus improving run-time performance. If the input data can be overwritten, the optimum memory usage can be achieved by also specifying the input array as the output array.

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `fft_size/2` twiddle factors. The function `twidfftrad2_fr16` may be used to initialize the array. If the twiddle table contains more factors than needed for a particular call on `cfft_fr16`, then the stride factor has to be set appropriately; otherwise it should be set to 1.

The arguments `block_exponent` and `scale_method` have been added for future expansion. These arguments are ignored by the function. To avoid overflow, the function scales the output by `1/fft_size`.

Algorithm

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

When the sequence length `n` is a power of 4, the `cfftrad4` algorithm is also available.

Domain

Input sequence length `n` must be a power of 2 and at least 8.

cfftf

fast N-point radix-4 complex input FFT

Synopsis

```
#include <filter.h>
void cfftf_fr16(const complex_fract16 input[],
                complex_fract16       output[],
                const complex_fract16 twiddle_table[],
                int                   twiddle_stride,
                int                   fft_size);
```

Description

The `cfftf_fr16` function transforms the time domain complex input signal sequence to the frequency domain by using the accelerated version of the "Discrete Fourier Transform" known as a "Fast Fourier Transform" or FFT. It "decimates in frequency" using an optimized radix-4 algorithm.

The size of the input array `input` and the output array `output` is `fft_size` where `fft_size` represents the number of points in the FFT. The `cfftf_fr16` function has been designed for optimum performance and requires that the input array `input` be aligned on an address boundary that is a multiple of four times the FFT size. For certain applications, this alignment constraint may not be appropriate; in such cases, the application should call the `cfftrad4_fr16` function ("cfftrad4" on page 4-71) instead, with no loss of facility (apart from performance).

The number of points in the FFT, fft_size, must be a power of 4 and must be at least 16.

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `3*fft_size/4` complex twiddle factors. The table should be initialized with complex twiddle factors in which the real coefficients are positive cosine values and the imaginary coefficients are negative sine values. The function `twidfftf_fr16` (see on page 4-156) may be used to initialize the array. If the twiddle table contains more factors than required for a particular FFT size, then the stride factor `twiddle_stride` has to be set appropriately; otherwise it should be set to 1.

It is recommended that the output array not be allocated in the same 4K memory sub-bank as either the input array or the twiddle table, as the performance of the function may otherwise degrade due to data bank collisions.

The function uses static scaling of intermediate results to prevent overflow and the final output therefore is scaled by `1/fft_size`.

> This library function makes use of the `M3` register. The `M3` register may be used by an emulator for context switching. Refer to the appropriate emulator documentation.

Algorithm

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

The `cfft_fr16` function (see "cfft" on page 4-66), which uses a radix-2 algorithm, must be used when the FFT size, `n`, is only a power of 2.

Domain

The number of points in the FFT must be a power of 4 and must be at least 16.

Example

```
#include <filter.h>

#define FFTSIZE 64

#pragma align 256
```

```
segment ("seg_1") complex_fract16 input[FFTSIZE];

#pragma align 4
segment ("seg_2") complex_fract16 output[FFTSIZE];

#pragma align 4
segment ("seg_3") complex_fract16 twid[(3*FFTSIZE)/4];

twidfftf_fr16(twid,FFTSIZE);
cfftf_fr16(input,
           output,
           twid,1,FFTSIZE);
```

cfftrad4

n point radix-4 complex input FFT

Synopsis

```
#include <filter.h>
void cfftrad4_fr16 (const complex_fract16  input[],
                    complex_fract16        temp[],
                    complex_fract16        output[],
                    const complex_fract16  twiddle_table[],
                    int                    twiddle_stride,
                    int                    fft_size,
                    int                    block_exponent,
                    int                    scale_method);
```

Description

This function transforms the time domain complex input signal sequence to the frequency domain by using the radix-4 Fast Fourier Transform. The `cfftrad4_fr16` function “decimates in frequency” by the radix-4 FFT algorithm.

The size of the input array `input`, the output array `output`, and the temporary working buffer `temp` is `fft_size`, where `fft_size` represents the number of points in the FFT. Memory bank collisions, which have an adverse effect on run-time performance, may be avoided by allocating all input and working buffers to different memory banks. If the input data can be overwritten, the optimum memory usage can be achieved by also specifying the input array as the output array.

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `3*fft_size/4` twiddle coefficients. The function `twidfftrad4_fr16` may be used to initialize the array. If the twiddle table contains more coefficients than needed for a particular call on `cfftrad4_fr16`, then the stride factor has to be set appropriately; otherwise it should be set to 1.

The arguments `block_exponent` and `scale_method` have been added for future expansion. These arguments are ignored by the function. To avoid overflow, the function performs static scaling by dividing the input by `fft_size`.

Algorithm

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

When the sequence length, `n=fft_size`, is not a power of 4, the radix-2 method must be used. See "cfft" on page 4-66.

Domain

Input sequence length `fft_size` must be a power of 4 and at least 16.

cfft2d

n x n point 2-D complex input FFT

Synopsis

```
#include <filter.h>
void cfft2d_fr16(const complex_fract16  *input
                 complex_fract16        *temp,
                 complex_fract16        *output,
                 const complex_fract16   twiddle_table[],
                 int                     twiddle_stride,
                 int                     fft_size,
                 int                     block_exponent,
                 int                     scale_method);
```

Description

This function computes the two-dimensional Fast Fourier Transform (FFT) of the complex input matrix `input[fft_size][fft_size]` and stores the result to the complex output matrix `output[fft_size][fft_size]`.

The size of the input array `input`, the output array `output`, and the temporary working buffer `temp` is `fft_size*fft_size`, where `fft_size` represents the number of points in the FFT. Memory bank collisions, which have an adverse effect on run-time performance, may be avoided by allocating all input and working buffers to different memory banks. If the input data can be overwritten, the optimum memory usage can be achieved by also specifying the input array as the output array.

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `fft_size` twiddle factors. The function `twidfft2d_fr16` may be used to initialize the array. If the twiddle table contains more factors than needed for a particular call on `cfft2d_fr16`, then the stride factor has to be set appropriately; otherwise it should be set to 1.

The arguments `block_exponent` and `scale_method` have been added for future expansion. These arguments are ignored by the function. To avoid overflow, the function scales the output by `fft_size*fft_size`.

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,...,n-1}; j={0,1,2,...,n-1}; a=`input`; c=`output`; n=fft_size

Domain

Input sequence length `fft_size` must be a power of 2 and at least 16.

cfir

complex finite impulse response filter

Synopsis

```
#include <filter.h>

void cfir_fr16(const complex_fract16  input[],
               complex_fract16        output[],
               int                    length,
               cfir_state_fr16       *filter_state);
```

The function uses the following structure to maintain the state of the filter.

```
typedef struct
{
   int k;                     /* Number of coefficients  */
   complex_fract16 *h;        /* Filter coefficients     */
   complex_fract16 *d;        /* Start of delay line     */
   complex_fract16 *p;        /* Read/write pointer      */
}  cfir_state_fr16;
```

Description

The `cfir_fr16` function implements a complex finite impulse response (CFIR) filter. It generates the filtered response of the complex input data `input` and stores the result in the complex output vector `output`.

The function maintains the filter state in the structured variable `filter_state`, which must be declared and initialized before calling the function. The macro `cfir_init`, in the `filter.h` header file, is available to initialize the structure.

It is defined as:

```
#define cfir_init(state, coeffs, delay, ncoeffs) \

   (state).h = (coeffs);  \

   (state).d = (delay);   \

   (state).p = (delay);   \

   (state).k = (ncoeffs)
```

The characteristics of the filter (passband, stopband, and so on) are dependent upon the number of complex filter coefficients and their values. A pointer to the coefficients should be stored in `filter_state->h`, and `filter_state->k` should be set to the number of coefficients.

Each filter should have its own delay line which is a vector of type `complex_fract16` and whose length is equal to the number of coefficients. The vector should be cleared to zero before calling the function for the first time and should not otherwise be modified by the user program. The structure member `filter_state->d` should be set to the start of the delay line, and the function uses `filter_state->p` to keep track of its current position within the vector.

Algorithm

$$y(k) = \sum_{j=0}^{k-1} h(j) * x(i-j) \; for \, i = 0,1..n$$

where x=`input`; y=`output`; n=fft_size

Domain

–1.0 to +1.0

clip

clip

Synopsis

```
#include <math.h>

int clip (int parm1, int parm2);
long int lclip (long int parm1, long int parm2);
long long int llclip (long long int parm1,
                      long long int parm2);

float fclipf (float parm1, float parm2);
double fclip (double parm1, double parm2);
long double fclipd (long double parm1, long double parm2);

fract16 clip_fr16 (fract16 parm1, fract16 parm2);
```

Description

The clip functions return the first argument if it is less than the absolute value of the second argument; otherwise they return the absolute value of the second argument if the first is positive, or minus the absolute value if the first argument is negative.

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_long_double cmltd (complex_long_double a,
                           complex_long_double b);
complex_fract16 cmlt_fr16 (complex_fract16 a, complex_fract16 b);
```

Description

The cmlt functions compute the complex multiplication of two complex inputs, `a` and `b`, and return the result.

Algorithm

```
Re(c) = Re(a) * Re(b) - Im(a) * Im(b)
Im(c) = Re(a) * Im(b) + Im(a) * Re(b)
```

Domain

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

-1.7×10^{308} to $+1.7 \times 10^{308}$ for `cmltd( )`

-1.0 to 1.0 for `cmlt_fr16( )`

coeff_iirdf1

convert coefficients for DF1 IIR filter

Synopsis

```
#include <filter.h>
void coeff_iirdf1_fr16 (const float acoeff[ ],
                        const float bcoeff[ ],
                        fract16 coeff[ ], int nstages);
```

Description

The `coeff_iirdf1_fr16` function transforms a set of A-coefficients and a set of B-coefficients into a set of coefficients for the `iirdf1_fr16` function (see on page 4-156), which implements an optimized, direct form 1 infinite impulse response (IIR) filter.

The `A`-coefficients and the `B`-coefficients are passed into the function via the floating-point vectors `acoeff` and `bcoeff`, respectively. The `A0` coefficients are assumed to be 1.0, and all other `A`-coefficients must be scaled according; the `A0` coefficients should not be included in the vector `acoeffs`. The number of stages in the filter is given by the parameter `nstages`, and therefore the size of the `acoeffs` vector is `2*nstages` and the size of the `bcoeffs` vector is `(2*nstages) + 1`.

The values of the coefficients that are held in the vectors `acoeffs` and `bcoeffs` must be in the range of `[LONG_MIN, LONG_MAX]`, that is they must not be less than -2147483648, or greater than 2147483647.

The `coeff_iirdf1_fr16` function scales the coefficients and stores them in the vector `coeff`. The function also stores the appropriate scaling factor in the vector which the `iirdf1_fr16` function will then apply to the filtered response that it generates (thus eliminating the need to scale the output generated by the IIR function). The size of `coeffs` array should be `(4*nstages) + 2`.

Algorithm

The `A`-coefficients and the `B`-coefficients represent the numerator and denominator coefficients of `H(z)`, where `H(z)` is defined as:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \ldots + b_{m+1} z^{-m}}{a_1 + a_2 z^{-1} + \ldots + a_{m+1} z^{-m}}$$

If any of the coefficients are greater than `0.999969` (the largest floating-point value that can be converted to a value of type `fract16`), then all the `A`-coefficients and all the `B`-coefficients are scaled to be less than 1.0. The coefficients are stored into the vector `coeffs` in the following order:

[b_0 , $-a_0 1$, $b_0 1$, $-a_0 2$, $b_0 2$, ... ,

$-a_n 1$, $b_n 1$, $-a_n 2$, $b_n 2$, `scale factor`]

where `n` is the number of stages.

Note that the `A`-coefficients are negated by the function.

Domain

`acoeff`, `bcoeff` = `[LONG_MIN, LONG_MAX]` where `LONG_MIN` and `LONG_MAX` are macros that are defined in the `limits.h` header file

conj

complex conjugate

Synopsis

```
#include <complex.h>
complex_float conjf (complex_float a);
complex_double conj (complex_double a);
complex_long_double conjd (complex_long_double a);
complex_fract16 conj_fr16 (complex_fract16 a);
```

Description

The conj functions conjugate the complex input `a` and return the result.

Algorithm

```
Re(c) = Re(a)
Im(c) = -Im(a)
```

Domain

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

-1.7×10^{308} to $+1.7 \times 10^{308}$ for `conjd( )`

–1.0 to 1.0 for `conj_fr16( )`

convolve

convolution

Synopsis

```
#include <filter.h>
void convolve_fr16(const fract16  input_x[],
                   int            length_x,
                   const fract16  input_y[],
                   int            length_y,
                   fract16        output[]);
```

Description

This function convolves two sequences pointed to by `input_x` and `input_y`. If `input_x` points to the sequence whose length is `length_x` and `input_y` points to the sequence whose length is `length_y`, the resulting sequence pointed to by `output` has length `length_x + length_y - 1`.

Algorithm

Convolution between two sequences `input_x` and `input_y` 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 + clen2-2`.
(Values for `cin1[j]` are considered to be zero for `j < 0` or `j > clen1-1`).
where cin1 =`input_x`
cin2 =`input_y`
cout =`output`
clen1=`length_x`
clen2=`length_y`

Example

Here is an example of a convolution where input_x is of length 4 and input_y is of length 3. If we represent input_x as “A” and input_y 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(const fract16  *input_x,
                 int              rows_x,
                 int              columns_x,
                 const fract16  *input_y,
                 int              rows_y,
                 int              columns_y,
                 fract16        *output);
```

Description

The conv2d function computes the two-dimensional convolution of input matrix input_x of size rows_x*columns_x and input_y of size rows_y*columns_y and stores the result in matrix output of dimension (rows_x + rows_y-1) x (columns_x + columns_y-1).

A temporary work area is allocated from the run-time stack that the function uses to preserve accuracy while evaluating the algorithm. The stack may therefore overflow if the sizes of the input matrices are sufficiently large. The size of the stack may be adjusted by making appropriate changes to the .LDF file

Algorithm

The two-dimensional convolution of min1[mrow1][mcol1] and min2[mrow2][mcol2] is defined as:

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

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

Domain

–1.0 to +1.0

conv2d3x3

2-D convolution with 3 x 3 matrix

Synopsis

```
#include <filter.h>
void conv2d3x3_fr16(const fract16  *input_x,
                    int             rows_x,
                    int             columns_x,
                    const fract16  *input_y,
                    fract16        *output);
```

Description

The conv2d3x3 function computes the two-dimensional circular convolution of matrix input_x (size [rows_x][columns_x]) with matrix input_y (size [3][3]).

Algorithm

Two-dimensional input matrix input_x is convolved with input matrix input_y, placing the result in a matrix pointed to by output.

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

for c = 0 to mcol1+2 and r = 0 to mrow1+2, where min1=input_x; min2=input_y; mcol1=columns_x; mrow1=rows_x; mout=output

Domain

–1.0 to +1.0

copysign

copysign

Synopsis

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

Description

The copysign functions copy the sign of the second argument to the first argument.

Algorithm

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

Domain

Full range for type of parameters used.

cot

cotangent

Synopsis

```
#include <math.h>
float cotf (float a);
double cot (double a);
long double cotd (long double a);
```

Description

These functions calculate the cotangent of their argument `a`, which is measured in radians. If `a` is outside of the domain, the functions return 0.

Algorithm

```
c = cot(a)
```

Domain

x = [–9099 ... 9099]   for `cotf( )`

x = [-4.21657e8 ... 4.21657e8]   for `cotd( )`

countones

count one bits in word

Synopsis

```
#include <math.h>
int countones(int parm);
int lcountones(long parm);
int llcountones(long long int parm);
```

Description

The countones functions count the number of one bits in the argument `parm`.

Algorithm

$$return = \sum_{j=0}^{N-1} bit[j] \;\; of \;\; parm$$

where `N` is the number of bits in `parm`.

crosscoh

cross-coherence

Synopsis

```
#include <stats.h>

void crosscohf (const float  samples_x[ ],
                const float  samples_y[ ],
                int          sample_length,
                int          lags,
                float        coherence[ ]);

void crosscoh (const double  samples_x[ ],
               const double  samples_y[ ],
               int           sample_length,
               int           lags,
               double        coherence[ ]);

void crosscohd (const long double  samples_x[ ],
                const long double  samples_y[ ],
                int                sample_length,
                int                lags,
                long double        coherence[ ]);

void crosscoh_fr16 (const fract16  samples_x[ ],
                    const fract16  samples_y[ ],
                    int            sample_length,
                    int            lags,
                    fract16        coherence[ ]);
```

Description

The crosscoh functions compute the cross-coherence of two input vectors `samples_x[]` and `samples_y[]`. The cross-coherence is the cross-correlation minus the product of the mean of `samples_x` and the mean of `samples_y`. The length of the input vectors is given by `sample_length`. The functions return the result in the array `coherence` with `lags` elements.

Algorithm

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

where k = {0,1,...,`lags`-1}; a=`samples_x`; b=`samples_y`; c=`coherence`; `a` is the mean value of input vector `a`; `b` is the mean value of input vector `b`.

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$	for `crosscohf ( )`
-1.7×10^{308} to $+1.7 \times 10^{308}$	for `crosscohd ( )`
-1.0 to $+1.0$	for `crosscoh_fr16 ( )`

crosscorr

cross-correlation

Synopsis

```
#include <stats.h>

void crosscorrf (const float  samples_x[ ],
                 const float  samples_y[ ],
                 int          sample_length,
                 int          lags,
                 float        correlation[ ]);

void crosscorr (const double  samples_x[ ],
                const double  samples_y[ ],
                int           sample_length,
                int           lags,
                double        correlation[ ]);

void crosscorrd (const long double  samples_x[ ],
                 const long double  samples_y[ ],
                 int                sample_length,
                 int                lags,
                 long double        correlation[ ]);

void crosscorr_fr16 (const fract16  samples_x[ ],
                     const fract16  samples_y[ ],
                     int            sample_length,
                     int            lags,
                     fract16        correlation[ ]);
```

Description

The crosscorr functions perform a cross-correlation between two signals. The cross-correlation is the sum of the scalar products of the signals in which the signals are displaced in time with respect to one another. The signals to be correlated are given by the input vectors `samples_x[]` and `samples_y[]`. The length of the input vectors is given by `sample_length`. The functions return the result in the array `correlation` with `lags` elements.

Cross-correlation is used in signal processing applications such as speech analysis.

Algorithm

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

where k = {0,1,...,`lags`-1}; a=`samples_x`; b=`samples_y`; n=`sample_length`

Domain

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

-1.7×10^{308} to $+1.7 \times 10^{308}$ for `csubd ( )`

-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_long_double csubd (complex_long_double a,
                           complex_long_double b);
complex_fract16 csub_fr16 (complex_fract16 a, complex_fract16 b);
```

Description

The csub functions compute the complex subtraction of two complex inputs, `a` and `b`, and return the result.

Algorithm

```
Re(c) = Re(a) - Re(b)
Im(c) = Im(a) - Im(b)
```

Domain

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

-1.7×10^{308} to $+1.7 \times 10^{308}$ for `csubd ( )`

-1.0 to 1.0 for `csub_fr16 ( )`

fir

finite impulse response filter

Synopsis

```
#include <filter.h>

void fir_fr16(const fract16    input[],
              fract16          output[],
              int              length,
              fir_state_fr16  *filter_state);
```

The 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;              /* number of coefficients           */
   int l;              /* interpolation/decimation index   */
} fir_state_fr16;
```

Description

The `fir_fr16` function implements a finite impulse response (FIR) filter. The function generates the filtered response of the input data `input` and stores the result in the output vector `output`. The number of input samples and the length of the output vector are specified by the argument `length`.

The function maintains the filter state in the structured variable `filter_state`, which must be declared and initialized before calling the function. The macro `fir_init`, defined in the `filter.h` header file, is available to initialize the structure.

It is defined as:

```
#define fir_init(state, coeffs, delay, ncoeffs. index) \
   (state).h = (coeffs);   \
   (state).d = (delay);    \
   (state).p = (delay);    \
   (state).k = (ncoeffs);  \
   (state).l = (index)
```

The characteristics of the filter (passband, stopband, and so on) are dependent upon the number of filter coefficients and their values. A pointer to the coefficients should be stored in `filter_state->h`, and `filter_state->k` should be set to the number of coefficients.

Each filter should have its own delay line which is a vector of type `fract16` and whose length is equal to the number of coefficients. The vector should be initially cleared to zero and should not otherwise be modified by the user program. The structure member `filter_state->d` should be set to the start of the delay line, and the function uses `filter_state->p` to keep track of its current position within the vector.

The structure member `filter_state->l` is not used by `fir_fr16`. This field is normally set to an interpolation/decimation index before calling either the `fir_interp_fr16` or `fir_decima_fr16` functions.

Algorithm

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

where x=`input`; y=`output`

Domain

–1.0 to +1.0

fir_decima

FIR decimation filter

Synopsis

```
#include <filter.h>

void fir_decima_fr16(const fract16    input[],
                     fract16          output[],
                     int              length,
                     fir_state_fr16  *filter_state);
```

The 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;               /* number of coefficients          */
   int l;               /* interpolation/decimation index  */
} fir_state_fr16;
```

Description

The `fir_decima_fr16` function performs an FIR-based decimation filter. It generates the filtered decimated response of the input data `input` and stores the result in the output vector `output`. The number of input samples is specified by the argument `length`, and the size of the output vector should be `length/l` where `l` is the decimation index.

The function maintains the filter state in the structured variable `filter_state`, which must be declared and initialized before calling the function. The macro `fir_init`, defined in the `filter.h` header file, is available to initialize the structure.

It is defined as:

```
#define fir_init(state, coeffs, delay, ncoeffs, index) \
   (state).h = (coeffs);   \
   (state).d = (delay);    \
   (state).p = (delay);    \
   (state).k = (ncoeffs);  \
   (state).l = (index)
```

The characteristics of the filter are dependent upon the number of filter coefficients and their values, and on the decimation index supplied by the calling program. A pointer to the coefficients should be stored in `filter_state->h`, and `filter_state->k` should be set to the number of coefficients. The decimation index is supplied to the function in `filter_state->l`.

Each filter should have its own delay line which is a vector of type `fract16` and whose length is equal to the number of coefficients. The vector should be initially cleared to zero and should not otherwise be modified by the user program. The structure member `filter_state->d` should be set to the start of the delay line, and the function uses `filter_state->p` to keep track of its current position within the vector.

Algorithm

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

where i = 0,1,...,(n/l) - 1; x=`input`; y=`output`

Domain

–1.0 to + 1.0

fir_interp

FIR interpolation filter

Synopsis

```
#include <filter.h>
void fir_interp_fr16(const fract16    input[],
                     fract16          output[],
                     int              length,
                     fir_state_fr16  *filter_state);
```

The 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;             /* number of coefficients per polyphase */
   int l;             /* interpolation/decimation index      */
} fir_state_fr16;
```

Description

The `fir_interp_fr16` function performs an FIR-based interpolation filter. It generates the interpolated filtered response of the input data `input` and stores the result in the output vector `output`. The number of input samples is specified by the argument `length`, and the size of the output vector should be `length*l` where `l` is the interpolation index.

The filter characteristics are dependent upon the number of polyphase filter coefficients and their values, and on the interpolation factor supplied by the calling program.

The `fir_interp_fr16` function assumes that the coefficients are stored in the following order:

```
coeffs[(np * ncoeffs) + nc]
```

where: `np = {0, 1, ..., nphases-1}`

`nc = {0, 1, ..., ncoeffs-1}`

In the above syntax, `nphases` is the number of polyphases and `ncoeffs` is the number of coefficients per polyphase. A pointer to the coefficients is passed into the `fir_interp_fr16` function via the argument `filter_state`, which is a structured variable that represents the filter state. This structured variable must be declared and initialized before calling the function. The `filter.h` header file contains the macro `fir_init` that can be used to initialize the structure and is defined as:

```
#define fir_init(state, coeffs, delay, ncoeffs, index) \
   (state).h = (coeffs);  \
   (state).d = (delay);   \
   (state).p = (delay);   \
   (state).k = (ncoeffs); \
   (state).l = (index)
```

The interpolation factor is supplied to the function in `filter_state->l`. A pointer to the coefficients should be stored in `filter_state->h`, and `filter_state->k` should be set to the number of coefficients per polyphase filter.

Each filter should have its own delay line which is a vector of type `fract16` and whose length is equal to the number of coefficients in each polyphase. The vector should be cleared to zero before calling the function for the first time and should not otherwise be modified by the user program. The structure member `filter_state->d` should be set to the start of the delay line, and the function uses `filter_state->p` to keep track of its current position within the vector.

Algorithm

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

where i = 0,1,...,n-1; m = 0,1,...,l-1; x=input; y=output

Domain

–1.0 to +1.0

Example

```
#include <filter.h>

#define INTERP_FACTOR         5
#define NSAMPLES             50
#define TOTAL_COEFFS         35
#define NPOLY INTERP_FACTOR
#define NCOEFFS (TOTAL_COEFFS/NPOLY)

/* Coefficients */

fract16 coeffs[TOTAL_COEFFS];

/* Input, Output, Delay Line, and Filter State */

fract16 input[NSAMPLES], output[INTERP_FACTOR*NSAMPLES];
fract16 delay[NCOEFFS];

fir_state state;

/* Utility Variables */

int i;
```

```
/* Initialize the delay line */

for (i = 0; i < NCOEFFS; i++)
    delay[i] = 0;

/* Initialize the filter state */

fir_init (state, coeffs, delay, NCOEFFS, INTERP_FACTOR);

/* Call the fir_interp_fr16 function */

fir_interp_fr16 (input, output, NSAMPLES, &state);
```

gen_bartlett

generate Bartlett window

Synopsis

```
#include <window.h>
void gen_bartlett_fr16(fract16  bartlett_window[],
                       int      window_stride,
                       int      window_size);
```

Description

This function generates a vector containing the Bartlett window. The length of the window required is specified by the parameter `window_size`, and the parameter `window_stride` is used to space the window values within the output vector `bartlett_window`. The length of the output vector should therefore be `window_size*window_stride`.

The Bartlett window is similar to the Triangle window (see on page 4-114) but has the following different properties:

- 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 an `N+2` Bartlett window equals an `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 w=`bartlett_window`; N=`window_size`; n = {0, 1, 2, ..., N-1}

Domain

window_stride > 0; N > 0

gen_blackman

generate Blackman window

Synopsis

```
#include <window.h>
void gen_blackman_fr16(fract16  blackman_window[],
                       int      window_stride,
                       int      window_size);
```

Description

This function generates a vector containing the Blackman window. The length of the window required is specified by the parameter `window_size`, and the parameter `window_stride` is used to space the window values within the output vector `blackman_window`. The length of the output vector should therefore be `window_size*window_stride`.

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=`window_size`; w= `blackman_window`; n = {0, 1, 2, ..., N-1}

Domain

`window_stride` > 0; N > 0

gen_gaussian

generate Gaussian window

Synopsis

```
#include <window.h>
void gen_gaussian_fr16(fract16  gaussian_window[],
                       float    alpha,
                       int      window_stride,
                       int      window_size);
```

Description

This function generates a vector containing the Gaussian window. The length of the window required is specified by the parameter `window_size`, and the parameter `window_stride` is used to space the window values within the output vector gaussian_window. The length of the output vector should therefore be `window_size*window_stride`.

The parameter `alpha` is used to control the shape of the window. In general, the peak of the Gaussian window will become narrower and the leading and trailing edges will tend towards zero the larger that `alpha` becomes. Conversely, the peak will get wider and wider the more that `alpha` tends towards zero.

Algorithm

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

where w=`gaussian_window`; N=`window_size`; n= {0, 1, 2, ..., N-1}; α is an input parameter.

Domain

`window_stride` > 0; `window_size` > 0; $\alpha > 0.0$

gen_hamming

generate Hamming window

Synopsis

```
#include <window.h>
void gen_hamming_fr16(fract16  hamming_window[],
                      int      window_stride,
                      int      window_size);
```

Description

This function generates a vector containing the Hamming window. The length of the window required is specified by the parameter `window_size`, and the parameter `window_stride` is used to space the window values within the output vector `hamming_window`. The length of the output vector should therefore be `window_size*window_stride`.

Algorithm

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

where w=`hamming_window`; N=`window_size`; n= {0, 1, 2, ..., N-1}

Domain

`window_stride` > 0; N > 0

gen_hanning

generate Hanning window

Synopsis

```
#include <window.h>
void gen_hanning_fr16(fract16  hanning_window[],
                     int      window_stride,
                     int      window_size);
```

Description

This function generates a vector containing the Hanning window. The length of the window required is specified by the parameter `window_size`, and the parameter `window_stride` is used to space the window values within the output vector `hanning_window`. The length of the output vector should therefore be `window_size*window_stride`. 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=`window_size`; w=`hanning_window`; n = {0, 1, 2, ..., N-1}

Domain

`window_stride` > 0; N > 0

gen_harris

generate Harris window

Synopsis

```
#include <window.h>
void gen_harris_fr16(fract16  harris_window[],
                     int      window_stride,
                     int      window_size);
```

Description

This function generates a vector containing the Harris window. The length of the window required is specified by the parameter `window_size`, and the parameter `window_stride` is used to space the window values within the output vector `harris_window`. The length of the output vector should therefore be `window_size*window_stride`. 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=`window_size`; w=`harris_window`; n = {0, 1, 2, ..., N-1}

Domain

`window_stride` > 0; N > 0

gen_kaiser

generate Kaiser window

Synopsis

```
#include <window.h>
void gen_kaiser_fr16(fract16  kaiser_window[],
                     float    beta,
                     int      window_stride,
                     int      window_size);
```

Description

This function generates a vector containing the Kaiser window. The length of the window required is specified by the parameter `window_size`, and the parameter `window_stride` is used to space the window values within the output vector `kaiser_window`. The length of the output vector should therefore be `window_size*window_stride`. The β value is specified by parameter `beta`.

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=`window_size`; w=`kaiser_window`; n = {0, 1, 2, ..., N-1}; α = (N - 1) / 2; I0(β) represents the zeroth-order modified Bessel function of the first kind.

Domain

a > 0; N > 0; β > 0.0

gen_rectangular

generate rectangular window

Synopsis

```
#include <window.h>
void gen_rectangular_fr16(fract16  rectangular_window[],
                          int      window_stride,
                          int      window_size);
```

Description

This function generates a vector containing the Rectangular window. The length of the window required is specified by the parameter `window_size`, and the parameter `window_stride` is used to space the window values within the output vector `rectangular_window`. The length of the output vector should therefore be `window_size*window_stride`.

Algorithm

rectangular_window[n] = 1

where N=`window_size`; n = {0, 1, 2, ..., N-1}

Domain

`window_stride` > 0; N > 0

gen_triangle

generate triangle window

Synopsis

```
#include <window.h>
void gen_triangle_fr16(fract16  triangle_window[],
                       int      window_stride,
                       int      window_size);
```

Description

This function generates a vector containing the Triangle window. The length of the window required is specified by the parameter `window_size`, and the parameter `window_stride` is used to space the window values within the output vector `triangle_window`.

Refer to the Bartlett window (on page 4-104) regarding the relationship between it and the Triangle window.

Algorithm

For even `n`, the following equation applies.

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

where N=`window_size`; w=`triangle_window`; n = {0, 1, 2, ..., N-1}

For odd `n`, the following equation applies.

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

where n = {0, 1, 2, ..., N-1}

Domain

`window_stride` > 0; N > 0

gen_vonhann

generate Von Hann window

Synopsis

```
#include <window.h>
void gen_vonhann_fr16(fract16  vonhann_window[],
                      int      window_stride,
                      int      window_size);
```

Description

This function is identical to the Hanning window (see on page 4-110).

Domain

`window_stride` > 0; `window_size` > 0

histogram

histogram

Synopsis

```
#include <stats.h>

void histogramf (const float  samples[],
                 int          histogram[],
                 float        max_sample,
                 float        min_sample,
                 int          sample_length,
                 int          bin_count);

void histogram (const double  samples[]
                int           histogram[],
                double        max_sample,
                double        min_sample,
                int           sample_length,
                int           bin_count);

void histogramd (const long double  samples[],
                 int                histogram[],
                 long double        max_sample,
                 long double        min_sample
                 int                sample_length,
                 int                bin_count);

void histogram_fr16 (const fract16  samples[],
                     int            histogram[],
                     fract16        max_sample,
                     fract16        min_sample,
                     int            sample_length,
                     int            bin_count);
```

Description

The histogram functions compute a histogram of the input vector `samples[ ]` that contains `nsamples` samples, and store the result in the output vector `histogram`.

The minimum and maximum value of any input sample is specified by `min_sample` and `max_sample`, respectively. These values are used by the function to calculate the size of each bin as `(max_sample – min_sample) / bin_count`, where `bin_count` is the size of the output vector `histogram`.

Any input value that is outside the range `[ min_sample, max_sample )` exceeds the boundaries of the output vector and is discarded.

> ⓘ To preserve maximum performance while performing out-of-bounds checking, the `histogram_fr16` function allocates a temporary work area on the stack. The work area is allocated with `(bin_count + 2)` elements and the stack may therefore overflow if the number of bins is sufficiently large. The size of the stack may be adjusted by making appropriate changes to the `.LDF` file.

Algorithm

Each input value is adjusted by `min_sample`, multiplied by `1/sample_length`, and rounded. The appropriate bin in the output vector is then incremented.

Domain

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

-1.7×10^{308} to $+1.7 \times 10^{308}$ for `histogramd ( )`

-1.0 to $+1.0$ for `histogram_fr16 ( )`

ifft

N-point radix-2 inverse FFT

Synopsis

```
#include <filter.h>
void ifft_fr16(const complex_fract16  input[],
               complex_fract16        temp[],
               complex_fract16        output[],
               const complex_fract16  twiddle_table[],
               int                    twiddle_size,
               int                    fft_size,
               int                    block_exponent
               int                    scale_method);
```

Description

This function transforms the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform.

The size of the input array `input`, the output array `output`, and the temporary working buffer `temp` is `fft_size`, where `fft_size` represents the number of points in the FFT. To avoid potential data bank collisions the input and temporary buffers should be allocated in different memory banks; this results in improved run-time performance. If the input data can be overwritten, the optimum memory usage can be achieved by also specifying the input array as the output array.

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `fft_size/2` twiddle coefficients. The function `twidfftrad2_fr16` may be used to initialize the array. If the twiddle table contains more coefficients than needed for a particular call on `ifft_fr16`, then the stride factor has to be set appropriately; otherwise it should be set to 1.

The arguments `block_exponent` and `scale_method` have been added for future expansion. These arguments are ignored by the function. To avoid overflow the function scales the output by `1/fft_size`.

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, the function 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 `fft_size` must be a power of 2 and at least 8.

Example

```
/* Compute  IFFT( CFFT( X ) ) = X */
#include <filter.h>

#define N_FFT  64
complex_fract16   in[N_FFT];
complex_fract16   out_cfft[N_FFT];
complex_fract16   out_ifft[N_FFT];
complex_fract16   temp[N_FFT];
complex_fract16   twiddle[N_FFT/2];

void ifft_fr16_example(void)
{
   int i;
   /* Generate DC signal */
   for( i = 0; i < N_FFT; i++ )
```

```
    {
       in[i].re = 0x100;
       in[i].im = 0x0;
    }

    /* Populate twiddle table */
    twidfftrad2_fr16(twiddle, N_FFT);

    /* Compute Fast Fourier Transform */
    cfft_fr16(in, temp, out_cfft, twiddle, 1, N_FFT, 0, 0);

    /* Reverse static scaling applied by cfft_fr16() function
       Apply the shift operation before the call to the
       ifft_fr16() function only if all the values in out_cfft
       = 0x100. Otherwise, perform the shift operation after the
       ifft_fr16() function has been computed.
    */
    for( i = 0; i < N_FFT; i++ )
    {
       out_cfft[i].re = out_cfft[i].re << 6; /* log2(N_FFT) = 6 */
       out_cfft[i].im = out_cfft[i].im << 6;
    }

    /* Compute Inverse Fast Fourier Transform
       The output signal from the ifft function will be the same
       as the DC signal of magnitude 0x100 which was passed into
       the cfft function.
    */
    ifft_fr16(out_cfft, temp, out_ifft, twiddle, 1, N_FFT, 0, 0);
}
```

ifftrad4

N-point radix-4 inverse input FFT

Synopsis

```
#include <filter.h>
void ifftrad4_fr16(const complex_float   *input,
                   complex_fract16       *temp,
                   complex_fract16       *output,
                   const complex_fract16  twiddle_table[],
                   int                    twiddle_stride,
                   int                    fft_size,
                   int                    block_exponent,
                   int                    scale_method);
```

Description

This function transforms the frequency domain complex input signal sequence to the time domain by using the radix-4 Inverse Fast Fourier Transform.

The size of the input array `input`, the output array `output`, and the temporary working buffer `temp` is `fft_size`, where `fft_size` represents the number of points in the FFT. Memory bank collisions, which have an adverse effect on run-time performance, may be avoided by allocating all input and working buffers to different memory banks. If the input data can be overwritten, the optimum memory usage can be achieved by also specifying the input array as the output array.

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `3/4fft_size` twiddle factors. The function `twidfftrad4_fr16` may be used to initialize the array. If the twiddle table contains more factors than needed for a particular call on `ifftrad4_fr16`, then the stride factor has to be set appropriately; otherwise it should be set to 1.

The arguments `block_exponent` and `scale_method` have been added for future expansion. These arguments are ignored by the function. To avoid overflow, the function performs static scaling by first dividing the input by `fft_size`.

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, the function 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 `fft_size` must be a power of 4 and at least 16.

ifft2d

n x n point 2-D inverse input FFT

Synopsis

```
#include <filter.h>
void ifft2d_fr16(const complex_float   *input,
                 complex_fract16       *temp,
                 complex_fract16       *output,
                 const complex_fract16  twiddle_table[],
                 int                    twiddle_stride,
                 int                    fft_size,
                 int                    block_exponent,
                 int                    scale_method);
```

Description

This function computes a two-dimensional Inverse Fast Fourier Transform of the complex input matrix `input[fft_size][fft_size]` and stores the result to the complex output matrix `output[fft_size][fft_size]`.

The size of the input array `input`, the output array `output`, and the temporary working buffer `temp` is `fft_size*fft_size`, where `fft_size` represents the number of points in the FFT. Memory bank collisions, which have an adverse effect on run-time performance, may be avoided by allocating all input and working buffers to different memory banks. If the input data can be overwritten, the optimum memory usage can be achieved by also specifying the input array as the output array.

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `fft_size` twiddle factors. The function `twidfft2d_fr16` may be used to initialize the array. If the twiddle table contains more factors than needed for a particular call on `ifft2d_fr16`, then the stride factor has to be set appropriately; otherwise it should be set to 1.

The arguments `block_exponent` and `scale_method` have been added for future expansion. These arguments are ignored by the function. To avoid overflow the function performs static scaling by dividing the input by `fft_size*fft_size`.

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,...,n-1}; j={0,1,2,...,n-1}

Domain

Input sequence length `fft_size` must be a power of 2 and at least 16.

iir

infinite impulse response filter

Synopsis

```
#include <filter.h>
void iir_fr16(const fract16    input[],
              fract16          output[],
              int              length,
              iir_state_fr16  *filter_state);
```

The function uses the following structure to maintain the state of the filter.

```
typedef struct
{
   fract16 *c;         /* coefficients             */
   fract16 *d;         /* start of delay line      */
   int k;              /* number of biquad stages  */
} iir_state_fr16;
```

Description

The `iir_fr16` function implements a biquad, transposed direct form II, infinite impulse response (IIR) filter. It generates the filtered response of the input data `input` and stores the result in the output vector `output`. The number of input samples and the length of the output vector are specified by the argument `length`.

The function maintains the filter state in the structured variable `filter_state`, which must be declared and initialized before calling the function. The macro `iir_init`, defined in the `filter.h` header file, is available to initialize the structure and is defined as:

```
#define iir_init(state, coeffs, delay, stages) \
   (state).c = (coeffs);  \
```

```
    (state).d = (delay);   \
    (state).k = (stages)
```

The characteristics of the filter are dependent upon filter coefficients and the number of stages. Each stage has five coefficients which must be stored in the order A2, A1, B2, B1, and B0. The value of A0 is implied to be 1.0 and A1 and A2 should be scaled accordingly. This requires that the value of the A0 coefficient be greater than both A1 and A2 for all the stages. The function iirdf1_fr16 (see on page 4-128) implements a direct form I filter, and does not impose this requirement; however, it does assume that the A0 coefficients are 1.0.

A pointer to the coefficients should be stored in filter_state->c, and filter_state->k should be set to the number of stages.

Each filter should have its own delay line which is a vector of type fract16 and whose length is equal to twice the number of stages. The vector should be initially cleared to zero and should not otherwise be modified by the user program. The structure member filter_state->d should be set to the start of the delay line.

Algorithm

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

where

$$D_m = X_m - A_2 * D_{m-2} - A_1 * D_{m-1}$$
$$Y_m = B_2 * D_{m-2} + B_1 * D_{m-1} + B_0 * D_m$$

where m = {0, 1, 2, ..., length-1}

Domain

–1.0 to +1.0

iirdf1

direct form I impulse response filter

Synopsis

```
#include <filter.h>
void iirdf1_fr16(const fract16       input[],
                 fract16             output[],
                 int                 length,
                 iirdf1_fr16_state  *filter_state);
```

The function uses the following structure to maintain the state of the filter.

```
typedef struct
{
   fract16 *c;     /* coefficients                  */
   fract16 *d;     /* start of delay line           */
   fract16 *p;     /* read/write pointer            */
   int k;          /* 2*number of biquad stages + 1  */
} iirdf1_fr16_state;
```

Description

The `iirdf1_fr16` function implements a biquad, direct form I, infinite impulse response (IIR) filter. It generates the filtered response of the input data `input` and stores the result in the output vector `output`. The number of input samples and the length of the output vector is specified by the argument `length`.

The function maintains the filter state in the structured variable `filter_state`, which must be declared and initialized before calling the function. The macro `iirdf1_init`, defined in the `filter.h` header file, is available to initialize the structure.

The macro is defined as:

```
#define iirdf1_init(state, coeffs, delay, stages) \
   (state).c = (coeffs);    \
   (state).d = (delay);     \
   (state).p = (delay);     \
   (state).k = (2*(stages)+1)
```

The characteristics of the filter are dependent upon the filter coefficients and the number of biquad stages. The `A`-coefficients and the `B`-coefficients for each stage are stored in a vector that is addressed by the pointer `filter_state->`c. This vector should be generated by the `coeff_iirdf1_fr16` function (see "coeff_iirdf1" on page 4-79). The variable `filter_state->k` should be set to the expression `(2*stages) + 1`.

Both the `iirdf1_fr16` and `iir_fr16` functions assume that the value of the `A0` coefficients is 1.0, and that all other `A`-coefficients have been scaled according. For the `iir_fr16` function, this also implies that the value of the `A0` coefficient is greater than both the `A1` and `A2` for all stages. This restriction does not apply to the `iirdf1_fr16` function because the coefficients are specified as floating-point values to the `coeff_iirdf1_fr16` function.

Each filter should have its own delay line which is a vector of type `fract16` and whose length is equal to `(4 * stages) + 2`. The vector should be initially cleared to zero and should not otherwise be modified by the user program. The structure member `filter_state->d` should be set to the start of the delay line, and the function uses `filter_state->p` to keep track of its current position within the vector. For optimum performance, coefficient and state arrays should be allocated in separate memory blocks.

The `iirdf1_fr16` function will adjust the output by the scaling factor that was applied to the `A`-coefficients and the `B`-coefficients by the `coeff_iirfd1_fr16` function.

Algorithm

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

where:

V $\quad$ = B0 * x(i) + B_1 * x(i-1) + B_2 * x(i-2)

y(i) = V + A_1 * y(i-1) + A_2 * y(i-2)

where i = {0, 1, .., length-1}
x = input
y = output

Domain

-1.0 to +1.0

Example

```
#include <filter.h>

#define NSAMPLES 50
#define NSTAGES 2

/* Coefficients for the coeff_iirdf1_fr16 function */

const float a_coeffs[(2 * NSTAGES)] = { . . . };
const float b_coeffs[(2 * NSTAGES) + 1] = { . . . };

/* Coefficients for the iirdf1_fr16 function */

fract16 df1_coeffs[(4 * NSTAGES) + 2];

/* Input, Output, Delay Line, and Filter State */
```

```
fract16 input[NSAMPLES], output[NSAMPLES];
fract16 delay[(4 * NSTAGES) + 2];
iirdf1_fr16_state state;
int i;

/* Initialize filter description */

iirdf1_init (state,df1_coeffs,delay,NSTAGES);

/* Initialize the delay line */

for (i = 0; i < ((4 * NSTAGES) + 2); i++)
    delay[i] = 0;

/* Convert coefficients */

coeff_iirdf1_fr16 (a_coeffs,b_coeffs,df1_coeffs,NSTAGES);

/* Call the function */

iirdf1_fr16 (input,output,NSAMPLES,&state);
```

max

maximum

Synopsis

```
#include <math.h>

int max (int parm1, int parm2);
long int lmax (long int parm1, long int parm2);
long long int llmax (long long int parm1, long long int parm2);

float fmaxf (float parm1, float parm2);
double fmax (double parm1, double parm2);
long double fmaxd (long double parm1, long double parm2);

fract16 max_fr16 (fract16 parm1, fract16 parm2);
```

Description

These functions return the larger of their two arguments.

Algorithm

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

Domain

Full range for type of parameters.

mean

mean

Synopsis

```
#include <stats.h>

float meanf(const float  samples[],
            int          sample_length);

double mean(const double  samples[],
            int           sample_length);

long double meand(const long double  samples[],
                  int                sample_length);

fract16 mean_fr16(const fract16  samples[],
                  int            sample_length);
```

Description

These functions return the mean of the input array `samples[ ]`. The number of elements in the array is sample_length.

Algorithm

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

Error Conditions

The `mean_fr16` function can be used to compute the mean of up to 65535 input data with a value of 0x8000 before the sum a_i saturates.

Domain

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

-1.7×10^{308} to $+1.7 \times 10^{308}$ for `meand ( )`

-1.0 to $+1.0$ for `mean_fr16 ( )`

min

minimum

Synopsis

```
#include <math.h>

int min (int parm1, int parm2);
long int lmin (long int parm1, long int parm2);
long long int llmin (long long int parm1, long long int parm2);

float fminf (float parm1, float parm2);
double fmin (double parm1, double parm2);
long double fmind (long double parm1, long double parm2);

fract16 min_fr16 (fract16 parm1, fract16 parm2);
```

Description

These functions return the smaller of their two 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(const short  input[],
                 short        output[],
                 int          length);
```

Description

This function takes a vector of linear 14-bit signed speech samples and performs μ-law compression according to ITU recommendation G.711. Each sample is compressed to 8 bits and is returned in the vector pointed to by `output`.

Algorithm

`C(k)= mu_law compression` of $A(k)$ for k = 0 to `length`-1

Domain

Content of input array: –8192 to 8191

mu_expand

μ-law expansion

Synopsis

```
#include <filter.h>
void mu_expand(const short  input[],
               short        output[],
               int          length);
```

Description

This function inputs a vector of 8-bit compressed speech samples and expands them according to ITU recommendation G.711. Each input value is expanded to a linear 14-bit signed sample in accordance with the μ-law definition and is returned in the vector pointed to `output`.

Algorithm

`C(k)= mu_law expansion` of A(*k*) for k = 0 to `length`-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);
complex_long_double normd (complex_long_double a);
```

Description

These functions normalize the complex input `a` and return the result.

Algorithm

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

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

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for normf ( )

-1.7×10^{308} to $+1.7 \times 10^{308}$ for `normd ( )`

polar

construct from polar coordinates

Synopsis

```
#include <complex.h>
complex_float polarf(float  magnitude,
                     float  phase);

complex_double polar(double  magnitude,
                     double  phase);

complex_long_double polard(long double  magnitude,
                           long double  phase);

complex_fract16 polar_fr16(fract16  magnitude,
                           fract16  phase);
```

Description

These functions transform the polar coordinate, specified by the arguments `magnitude` and `phase`, into a Cartesian coordinate and return the result as a complex number in which the x-axis is represented by the real part, and the y-axis by the imaginary part. The phase argument is interpreted as radians.

For the `polar_fr16` function, the phase must be scaled by 2π and must be in the range `[0x8000, 0x7ff0]`. The value of the phase may be either positive or negative. Positive values are interpreted as an anti-clockwise motion around a circle with a radius equal to the magnitude as shown in Table 4-10.

Table 4-11 shows how negative values for the phase argument are interpreted as a clockwise movement around a circle.

Table 4-10. Positive Phases for polar_fr16

Phase	Radians
0.0	0
0.25(0x2000)	π/2
0.50(0x4000)	π
0.75(0x6000)	3/2π
0.999(0x7ff0)	<2π

Table 4-11. Negative Phases for polar_fr16

Phase	Radians
-0.25(0xe000)	3/2π
-0.50(0xc000)	π
-0.75(0xa000)	π/2
-1.00(0x8000)	2 π

Algorithm

```
Re(c) = r*cos(θ)

Im(c) = r*sin(θ)
```

where θ is the phase; `r` is the magnitude

Domain

phase = [–4.3e7 … 4.3e7] for `polarf( )`

magnitude = $-3.4 \times 10^{38} \ldots +3.4 \times 10^{38}$ for `polarf( )`

phase = [–8.4331e8 … 8.4331e8] for `polard( )`

magnitude = -1.7×10^{308} to $+1.7 \times 10^{308}$ for `polard( )`

phase = [–1.0 ...+0.999969] for `polar_fr16( )`

magnitude = [–1.0 ... 1.0) for `polar_fr16( )`

Example

```
#include <complex.h>

#define PI 3.14159265

complex_fract16 point;
float phase_float;

fract16 phase_fr16;
fract16 mag_fr16;

phase_float = PI;
phase_fr16 = (phase_float / (2*PI)) * 32768.0;
mag_fr16 = 0x0200;

point = polar_fr16 (mag_fr16,phase_fr16);
        /* point.re = 0xfe00 */
        /* point.im = 0x0000 */
```

rfft

N-point radix-2 real input FFT

Synopsis

```
#include <filter.h>
void rfft_fr16(const fract16          input[],
               complex_fract16        temp[],
               complex_fract16        output[],
               const complex_fract16  twiddle_table[],
               int                    twiddle_stride,
               int                    fft_size,
               int                    block_exponent,
               int                    scale_method);
```

Description

This function transforms the time domain real input signal sequence to the frequency domain by using the radix-2 FFT. The function 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.

The size of the input array `input`, the output array `output`, and the temporary working buffer `temp` is `fft_size`, where `fft_size` represents the number of points in the FFT. Memory bank collisions, which have an adverse effect on run-time performance, may be avoided by allocating all input and working buffers to different memory banks. If the input data can be overwritten, the optimum memory usage can be achieved by also specifying the input array as the output array, provided that the memory size of the input array is at least `2*fft_size`.

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `fft_size/2` twiddle factors. The function `twidfftrad2_fr16` may be used to initialize the array. If the twiddle table contains more factors than needed for a particular call on `rfft_fr16`, then the stride factor has to be set appropriately; otherwise it should be set to 1.

The arguments `block_exponent` and `scale_method` have been added for future expansion. These arguments are ignored by the function. To avoid overflow, the function performs static scaling by dividing the input by `1/fft_size`.

Algorithm

See "cfft" on page 4-66 for more information.

Domain

Input sequence length `fft_size` must be a power of 2 and at least 8.

rfftrad4

N-point radix-4 real input FFT

Synopsis

```
#include <filter.h>
void rfftrad4_fr16(const fract16          input[],
                   complex_fract16        temp[],
                   complex_fract16        output[],
                   const complex_fract16  twiddle_table[],
                   int                    twiddle_stride,
                   int                    fft_size,
                   int                    block_exponent,
                   int                    scale_method);
```

Description

This function transforms the time domain real input signal sequence to the frequency domain by using the radix-4 Fast Fourier Transform. The `rfftrad4_fr16` function 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.

The size of the input array `input`, the output array `out`, and the temporary working buffer `temp` is `fft_size`, where `fft_size` represents the number of points in the FFT. To avoid potential data bank collisions, the input and temporary buffers should reside in different memory banks. This results in improved run-time performance. If the input data can be overwritten, the optimum memory usage can be achieved by also specifying the input array as the output array, provided that the memory size of the input array is at least `2*fft_size`.

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `3*fft_size/4` twiddle factors. The function `twidfftrad4_fr16` may be used to initialize the array. If the twiddle table contains more factors than needed for a particular call on rfftrad4_fr16, then the stride factor has to be set appropriately; otherwise it should be set to 1.

The arguments block_exponent and scale_method have been added for future expansion. These arguments are ignored by the function. To avoid overflow, the function performs static scaling by dividing the input by fft_size.

Algorithm

See "cfftrad4" on page 4-71 for more information.

Domain

Input sequence length fft_size must be a power of 4 and at least 16.

rfft2d

n x n point 2-D real input FFT

Synopsis

```
#include <filter.h>
void rfft2d_fr16(const fract16          input[],
                 complex_fract16        temp[],
                 complex_fract16        output[],
                 const complex_fract16  twiddle_table[],
                 int                    twiddle_stride,
                 int                    fft_size,
                 int                    block_exponent,
                 int                    scale_method);
```

Description

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

The size of the input array `input`, the output array `output`, and the temporary working buffer `temp` is `fft_size*fft_size`, where `fft_size` represents the number of points in the FFT. Improved run-time performance can be achieved by allocating the input and temporary arrays in separate memory banks; this avoids any memory bank collisions. If the input data can be overwritten, the optimum memory usage can be achieved by also specifying the input array as the output array, provided that the memory size of the input array is at least `2*fft_size*fft_size`.

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `fft_size` twiddle coefficients. The function `twidfft2d_fr16` may be used to initialize the array. If the twiddle table contains more coefficients than needed for a particular call on rfft2d_fr16, then the stride factor has to be set appropriately; otherwise it should be set to 1.

The arguments block_exponent and scale_method have been added for future expansion. These arguments are ignored by the function. To avoid overflow, the function scales the output by fft_size*fft_size.

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,...,n-1}; j={0,1,2,...,n-1}

Domain

Input sequence length fft_size must be a power of 2 and at least 16.

rms

root mean square

Synopsis

```
#include <stats.h>

float rmsf(const float  samples[],
           int          sample_length);

double rms(const double  samples[],
           int           sample_length);

long double rmsd(const long double  samples[],
                 int                sample_length);

fract16 rms_fr16(const fract16  samples[],
                 int            sample_length);
```

Description

These functions return the root mean square of the elements within the input vector `samples[ ]`. The number of elements in the vector is `sample_length`.

Algorithm

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

where a=`samples`; n=`sample_length`

Domain

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

-1.7×10^{308} to $+1.7 \times 10^{308}$ for `rmsd ( )`

-1.0 to $+1.0$ for `rms_fr16 ( )`

rsqrt

reciprocal square root

Synopsis

```
#include <math.h>
float rsqrtf (float a);
double rsqrt (double a);
long double rsqrtd (long double a);
```

Description

These functions calculate the reciprocal of the square root of the number `a`. If `a` is negative, the functions return 0.

Algorithm

$c = 1/\sqrt{a}$

Domain

0.0 ... 3.4 x 1038 for `rsqrtf ( )`

0.0 ... $+1.7 \times 10^{308}$ for `rsqrtd ( )`

twidfftrad2

generate FFT twiddle factors for radix-2 FFT

Synopsis

```
#include <filter.h>
void twidfftrad2_fr16(complex_fract16  twiddle_table[],
                      int              fft_size);
```

Description

This function calculates complex twiddle coefficients for a radix-2 FFT with `fft_size` points and returns the coefficients in the vector `twiddle_table`. The vector `twiddle_table`, known as the twiddle table, is normally calculated once and is then passed to an FFT function as a separate argument. The size of the table must be at least `1/2N`, where `N` is the number of points in the FFT.

FFTs of different sizes can be accommodated with the same twiddle 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 the FFT uses only half the points of the largest FFT, the stride should be 2 (this takes only every other element).

Algorithm

This function takes FFT length `fft_size` as an input parameter and generates the lookup table of complex twiddle coefficients. The samples are:

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

$$twid\_im(k) = -\sin\left(\frac{2\pi}{n}k\right)$$

where n=`fft_size`; k = {0, 1, 2, ..., n/2 - 1}

Domain

The FFT length `fft_size` must be a power of 2 and at least 8.

twidfftrad4

generate FFT twiddle factors for radix-4 FFT

Synopsis

```
#include <filter.h>
void twidfftrad4_fr16(complex_fract16  twiddle_table[],
                      int              fft_size);

void twidfft_fr16(complex_fract16  twiddle_table[],
                  int              fft_size);
```

Description

The `twidfftrad4_fr16` function initializes a table with complex twiddle factors for a radix-4 FFT. The number of points in the FFT are defined by `fft_size`, and the coefficients are returned in the twiddle table `twiddle_table`.

The size of the twiddle table must be at least `3*fft_size/4`, the length of the FFT input sequence. A table can accommodate several FFTs of different sizes by allocating the table at maximum size, and then using the stride argument of the FFT function to specify the step size through the table.

If the stride is set to 1, the FFT function uses all the table; if your FFT has only a quarter of the number of points of the largest FFT, the stride should be 4.

For efficiency, the twiddle table is normally generated once during program initialization and is then supplied to the FFT routine as a separate argument.

The `twidfft_fr16` routine is provided as an alternative to the `twidfftrad4_fr16` routine and performs the same function.

Algorithm

This function takes FFT length `fft_size` as an input parameter and generates the lookup table of complex twiddle coefficients.

The samples generated are:

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

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

where n=`fft_size`; k = {0, 1, 2, ..., ¾n - 1}

Domain

The FFT length `fft_size` must be a power of 4 and at least 16.

twidfftf_fr16

generate FFT twiddle factors for a fast FFT

Synopsis

```
#include <filter.h>
void twidfftf_fr16(complex_float  twiddle_table[ ],
                   int            fft_size);
```

Description

The `twidfftf_fr16` function generates complex twiddle factors for the fast radix-4 FFT function `cfftf_fr16` (on page 4-154), and stores the coefficients in the vector `twiddle_table`. The vector `twiddle_table`, known as the twiddle table, is normally calculated once and is then passed to the fast FFT as a separate argument. The size of the table must be `3/4N`, where `N` is the number of points in the FFT.

The same twiddle table may be used to calculate FFTs of different sizes provided that the table is generated for the largest FFT. Each FFT function has a stride parameter that the function uses to stride through the twiddle table. Normally, this stride parameter is set to 1, but to generate a smaller FFT, the argument should be scaled appropriately. For example, if a twiddle table is generated for an FFT with N points, then the same twiddle table may be used to generate a N/4-point FFT, provided that the stride parameter is set to 4, or a N/8-point FFT, if the parameter is set to 8.

The twiddle table generated by the `twidfftf_fr16` function is not compatible with the twiddle table generated by the `twidfftrad4_fr16` function (see on page 4-156).

Algorithm

The function calculates a lookup table of complex twiddle factors. The coefficients generated are:

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

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

where n=`fft_size`; k = {0, 1, 2, ..., 3/4n – 1}

Domain

The number of points in the FFT must be a power of 4 and must be at least 16.

twidfft2d

generate FFT twiddle factors for 2-D FFT

Synopsis

```
#include <filter.h>
void twidfft2d_fr16 (complex_fract16  twiddle_table[],
                     int              fft_size);
```

Description

The `twidfft2d_fr16` function generates complex twiddle factors for a 2-D FFT. The size of the FFT input sequence is given by the argument `fft_size` and the function writes the twiddle factors to the vector `twiddle_table`, known as the twiddle table.

The size of the twiddle table must be at least `fft_size`, the number of points in the FFT. Normally, the table is only calculated once and is then passed to an FFT function as an argument. A twiddle table may be used to generate several FFTs of different sizes by initializing the table for the largest FFT and then using the stride argument of the FFT function to specify the step size through the table. For example, to generate the largest FFT, the stride is set to 1, and to generate an FFT of half this size the stride is set to 2.

Algorithm

This function takes FFT length `fft_size` as an input parameter and generates the lookup table of complex twiddle coefficients.

The samples generated are:

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

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

where n=`fft_size`; k = {0, 1, 2, ..., n-1}

Domain

The FFT length `fft_size` must be a power of 2 and at least 16.

var

variance

Synopsis

```
#include <stats.h>

float varf(const float  samples[ ],
           int          sample_length);

double var(const double  samples[ ],
           int           sample_length);

long double vard(const long double  samples[ ],
                 int                sample_length);

fract16 var_fr16(const fract16  samples[ ],
                 int            sample_length);
```

Description

These functions return the variance of the elements within the input vector `samples[ ]`. The number of elements in the vector is `sample_length`.

Error Conditions

The `var_fr16` function can be used to compute the mean of up to 65535 input data with a value of 0x8000 before the sum a_i saturates.

Algorithm

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

where a=`samples`; n=`sample_length`

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$	for `varf( )`
-1.7×10^{308} to $+1.7 \times 10^{308}$	for `vard( )`
-1.0 to $+1.0$	for `var_fr16( )`

zero_cross

count zero crossings

Synopsis

```
#include <stats.h>

int zero_crossf (const float  samples[ ],
                 int          samples_length);

int zero_cross (const double  samples[ ],
                int           samples_length);

int zero_crossd (const long double  samples[ ],
                 int                samples_length);

int zero_cross_fr16 (fract16 samples[ ],
                     int     samples_length);
```

Description

The `zero_cross` functions return the number of times that a signal represented in the input array `samples[]` crosses over the zero line. If all the input values are either positive or zero, or they are all either negative or zero, then the functions return a zero.

Algorithm

The actual algorithm is different from the one shown below because the algorithm needs to handle the case where an element of the array is zero. However, this example gives you a basic understanding.

```
if ( a(i) > 0 && a(i+1) < 0 )|| (a(i) < 0 && a(i+1) > 0 )

      the number of zeros is increased by one
```

Domain

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

-1.7×10^{308} to $+1.7 \times 10^{308}$ for `zero_crossd ( )`

-1.0 to $+1.0$ for `zero_cross_fr16 ( )`