# 4 DSP LIBRARY FOR ADSP-2106x PROCESSORS

## Overview

The run-time library for ADSP-2106x processors contains a collection of functions that provide services commonly required by DSP applications; these functions are in addition to the C/C++ run-time library functions that are described in Chapter 3. The services provided by the DSP library functions include support for interrupt handling, signal processing, and access to hardware registers. All these services are Analog Devices extensions to ANSI standard C.

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

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

- "DSP Run-Time Library Guide" (starting on page 4-2) contains introductory information about the ADI special header files and built-in functions that are included with this release of the cc21k compiler.
- "DSP Run-Time Library Reference" (starting on page 4-13) contains the complete reference information for each DSP run-time library function included with this release of the cc21k compiler.

The C++ library reference information in HTML format is included on the software distribution CD-ROM. To access the reference files from VisualDSP++, see the procedure described in "Related Documents" on page 1-5. Select the *C++ Run-Time Library Reference* from the list of documents.

You can also manually access the HTML files using a web browser.

# DSP Run-Time Library Guide

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

- "Linking DSP Library Functions" on page 4-3
- "Working With Library Source Code" on page 4-3
- "DSP Header Files" on page 4-4
- "Built-In DSP Functions" on page 4-11

For information on the contents of the DSP library, see "DSP Run-Time Library Reference" on page 4-13 and on-line Help.

## Linking DSP Library Functions

When your C code calls a DSP run-time library function, the call creates a reference that the linker resolves when linking your program. One way to direct the linker to the location of the DSP library is to use the default Linker Description File (`ADSP-21<your_target>.ldf`). The default Linker Description File will automatically direct the linker to the library `libdsp.dlb` in the `21k\lib` subdirectory of your VisualDSP installation. If not using the default LDF file, then either add `libdsp.dlb` to the LDF used for your project, or alternatively use the compiler's `-ldsp` switch to specify that `libdsp.dlb` is to be added to the link line.

## Working With Library Source Code

The source code for the functions and macros in the DSP run-time library is provided with your VisualDSP software. By default, the installation program copies the source code to a subdirectory of the directory where the run-time libraries are kept, named `...\21k\lib\src`. The directory contains the source for the C run-time library, for the DSP run-time library, and for the I/O run-time library, as well as the source for the main program start-up functions. If you do not intend to modify any of the run-time library functions, you can delete this directory and its contents to conserve disk space.

The source code is provided so you can customize any particular function for your own needs. To modify these files, you need proficiency in ADSP-21xxx assembly language and an understanding of the run-time environment, as explained in "C/C++ Run-Time Model" (starting on page 2-121). Before you make any modifications to the source code, copy the source code to a file with a different filename and rename the function itself. Test the function before you use it in your system to verify that it is functionally correct. Note that Analog Devices supports the run-time library functions only as provided.

## DSP Header Files

The following DSP header files are supplied with this release of the cc21k compiler.

### 21060.h — ADSP-2106x DSP Functions

The `21060.h` header file includes the ADSP-2106x processor-specific functions of the DSP library, such as `poll_flag_in()`, `timer_set()`, and `idle()`. The `timer_set()`, `timer_on()`, and `timer_off()` functions are also available as in-line functions.

### 21065l.h — ADSP-21065L DSP Functions

The `21065l.h` header file includes the ADSP-21065L processor-specific functions of the DSP library, such as `poll_flag_in()` and `idle()`. The header file also includes support for the two programmable timers in the form of in-line functions.

### asm_sprt.h — Mixed C/Assembly Support

The `asm_sprt.h` header file consists of the ADSP-21xxx family assembly language macros, not C functions. They are used in your assembly routines that interface with C functions. For more information on this header file, see "Using Mixed C/C++ and Assembly Support Macros" on page 2-153.

### comm.h — A-law and µ-law Companders

The `comm.h` header file includes the voice-band compression and expansion communication functions of the DSP library.

### complex.h — Basic Complex Arithmetic Functions

The `complex.h` header file contains the type definition and some basic functions for `complex_float` variables.

The following structure is used to represent complex numbers in rectangular coordinates:

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

### def21060.h — ADSP-21060 Bit Definitions

The `def21060.h` header file includes macro definitions to enable usage of symbolic names for the system register bits for the ADSP-21060 processors. It also contains macro definitions for the IOP register addresses and bit fields.

### def21061.h — ADSP-21061 Bit Definitions

The `def21061.h` header file includes macro definitions to enable usage of symbolic names for the system register bits for the ADSP-21061 processors. It also contains macro definitions for the IOP register addresses and bit fields.

### def21062.h — ADSP-21062 Bit Definitions

The `def21062.h` header file includes macro definitions to enable usage of symbolic names for the system register bits for the ADSP-21062 processors. It also contains macro definitions for the IOP register addresses and bit fields.

### def21065l.h — ADSP-21065L Bit Definitions

The `def21065l.h` header file includes macro definitions to enable usage of symbolic names for the system register bits for the ADSP-21065L processors. It also contains macro definitions for the IOP register addresses and bit fields.

### dma.h — DMA Support Functions

The `dma.h` header file provides definitions and setup, status, enable and disable functions for DMA operations.

### filters.h — DSP Filters

The `filters.h` header file includes the digital signal processing filter functions of the DSP library.

### macros.h — Circular Buffers

The `macro.h` header file consists of ADSP-21xxx family assembly language macros, not C functions. Some are used to manipulate the circular buffer features of the ADSP-21xxx family processors.

### math.h — Math Functions

The standard math functions defined in `math.h` have been augmented by implementations for the `float` data type and some additional functions that are Analog Devices extensions to the ANSI standard. Table 4-1 provides a summary of the additional library functions defined by the `math.h` header file.

Table 4-1. Math Library - Additional Functions

| Description | Prototype |
|---|---|
| sign copy | `double copysign (double x, double y);`<br>`float copysignf (float x, float y);` |
| cotangent | `double cot (double x);`<br>`float cotf (float x);` |
| average | `double favg (double x, double y);`<br>`float favgf (float x, float y);` |
| clip | `double fclip (double x, double y);`<br>`float fclipf (float x, float y);` |
| maximum | `double fmax (double x, double y);`<br>`float fmaxf (float x, float y);` |
| minimum | `double fmin (double x, double y);`<br>`float fminf (float x, float y);` |
| reciprocal of square root | `double rsqrt (double x, double y);`<br>`float rsqrtf (float x, float y);` |

The following functions are only available if `double` is the same size as `float`:

`copysign`

`favg`

`fclip`

`fmax`

`fmin`

### matrix.h — Matrix Functions

The `matrix.h` header file contains functions for operating on real matrices; specifically it defines functions for adding and subtracting two matrices and for multiplying a matrix by either a scalar or a matrix.

### saturate.h — Saturation Mode Arithmetic

The `saturate.h` header file defines the interface for the saturated arithmetic operations. See "Saturated Arithmetic" on page 2-87 for further information.

### sport.h — Serial Port Support Functions

The `sport.h` header file provides definitions and setup, enable, and disable functions for the SHARC serial ports.

### stats.h — Statistical Functions

The `stats.h` header file includes various statistics functions of the DSP library, such as `mean()` and `autocorr()`.

### sysreg.h — Register Access

The `sysreg.h` header file defines a set of built-in functions that provide efficient access to the SHARC system registers from C. The supported functions are fully described in the section "Access to System Registers" in Chapter 2 Compiler.

### trans.h — Fast Fourier Transforms

The `trans.h` header file includes the Fast Fourier Transform functions of the DSP library. Note that `cfftN` stands for the entire family of Fast Fourier Transform functions `cfft65536`, `cfft32768`, etc.

The `cfftN` functions compute the fast Fourier transform (FFT) of their N-point complex input signal. The `ifftN` functions compute the inverse fast Fourier transform of their N-point complex input signal. The input to each of these functions is two float arrays (real and imaginary) of N elements. The routines output two N-element arrays.

If you only wish to input the real part of a signal, ensure that the imaginary input array is filled with zeros before calling the function.

The functions first bit-reverse the input arrays and then process them with an optimized block-floating-point FFT (or IFFT) routine.

The `rfftN` functions work like `cfftN` functions, except they operate on input arrays of real data only. This is equivalent to a `cfftN` whose imaginary input component is set to zero.

### 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 4-2.

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

Table 4-2. Window Generator Functions

| Description | Prototype |
|---|---|
| generate bartlett window | `void gen_bartlett (float w[], int a, int n)` |
| generate blackman window | `void gen_blackman (float w[], int a, int n)` |
| generate gaussian window | `void gen_gaussian (float w[], float alpha, int a, int n)` |
| generate hamming window | `void gen_hamming (float w[], int a, int n)` |
| generate hanning window | `void gen_hanning (float w[], int a, int n)` |
| generate harris window | `void gen_harris (float w[], int a, int n)` |
| generate kaiser window | `void gen_kaiser (float w[], float beta, int a, int n)` |
| generate rectangular window | `void gen_rectangular (float w[], int a, int n)` |
| generate triangle window | `void gen_triangle (float w[], int a, int n)` |

## Built-In DSP Functions

The C/C++ compiler supports built-in functions (also known as intrinsic functions) that enable efficient use of hardware resources. Knowledge of these functions is built into the compiler. Your program uses them via normal function call syntax. The compiler notices the invocation and replaces a call to a DSP library function with one or more machine instructions, just as it does for normal operators like "+" and "*".

Built-in functions are declared in system header files and have names which begin with double underscores, `__builtin`.

Identifiers beginning with "__" are reserved by the C standard, so these names do not conflict with user defined identifiers.

These functions are specific to individual architectures. The built-in DSP library functions supported at this time on the ADSP-2106x architectures are listed in Table 4-3. Refer to "Using the Compiler's Built-In C library Functions" on page 3-18 for further information on this topic.

Table 4-3. Built-in DSP Functions

| | |
|---|---|
| `copysign`[1] | `copysignf` |
| `favg`[1] | `favgf` |
| `fmax`[1] | `fmaxf` |
| `fmin`[1] | `fminf` |

[1] These functions will only be compiled as a built-in function if `double` is the same size as `float`.

Use the `-no-builtin` compiler switch to disable this feature.

The compiler also supports a set of built-in functions for which no line machine instructions are substituted. This set of built-in functions is characterized by defining one or more pointers in their argument list.

For this set of built-in functions, the compiler relaxes the normal rule whereby any pointer that is passed to a library function must address Data Memory (DM). The compiler recognizes when certain pointers address Program Memory (PM) and will generate a call to an appropriate version of the run-time library function. The following is a list of library functions that may be called with pointers that address Program Memory:

```
matadd

matmul

matscalmult

matsub
```

Use the `-no-builtin` compiler switch to disable this feature.

# DSP Run-Time Library Reference

The DSP run-time library is a collection of functions that you can call from your C programs. This section lists the functions in alphabetical order. Note the following items that apply to all the functions in the library.

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

Function Benchmarks and Specifications. All functions have been timed from setup, to invocation, to results storage of returned value. This includes all register storing, parameter passing, etc. Most functions execute slightly faster if you pass constants as arguments instead of variables.

Restrictions. When polymorphic functions are used and the function returns a pointer to program memory, cast the output of the function to pm. For example:

```
(char pm *)
```

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

*Name* and Purpose of the function

Synopsis—Required header file and functional prototype

Description—Function specification

Error Conditions—Method function uses to indicate an error

Example—Typical function usage

See Also—Related functions

## a_compress

A-law compression

### Synopsis

```
#include <comm.h>
int a_compress (int x);
```

### Description

The `a_compress` function takes a linear 13-bit signed speech sample and compresses it according to CCITT recommendation G.711. The value returned is an 8-bit sample that can be sent directly to an A-law codec.

### Error Conditions

The `a_compress` function does not return an error condition.

### Example

```
#include <comm.h>
int sample, compress;

compress = a_compress (sample);
```

### See Also

a_expand, mu_compress

## a_expand

A-law expansion

### Synopsis

```
#include <comm.h>
int a_expand (int compress_x);
```

### Description

The `a_expand` function takes an 8-bit compressed speech sample and expands it according to CCITT recommendation G.711 (A-law definition). The value returned is a linear 13-bit signed sample.

### Error Conditions

The `a_expand` function does not return an error condition.

### Example

```
#include <comm.h>
int compressed_sample, expanded;

expanded = a_expand (compressed_sample);
```

### See Also

a_compress, mu_expand

## autocoh

autocoherence

### Synopsis

```
#include <stats.h>
float *autocoh (float dm out[],
                const float dm in[],
                int samples,
                int lags);
```

### Description

The `autocoh` function computes the autocoherence of the floating-point input, `in[]`. The autocoherence of an input signal is its autocorrelation minus its mean. The function returns a pointer to the output array, `out[]` of length `lags`.

### Error Conditions

The `autocoh` function does not return an error condition.

### Example

```
#include <stats.h>
#define SAMPLES 1024

float excitation[SAMPLES], response[16];
int    lags = 16;

autocoh (response, excitation, SAMPLES, lags);
```

### See Also

autocorr, crosscoh, crosscorr

## autocorr

autocorrelation

### Synopsis

```
#include <stats.h>
float *autocorr (float dm out[],
                 const float dm in[],
                 int samples,
                 int lags);
```

### Description

The `autocorr` function performs 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 `in[]` 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 `samples`. This function returns a pointer to the `out[]` output data array of length `lags`.

The `autocorr` function is used in digital signal processing applications such as speech analysis.

### Error Conditions

The `autocorr` function does not return an error condition.

### Example

```
#include <stats.h>
float r[10], s[160];

autocorr (r, s, 160, 10);
    /* compute first 10 autocorr coefficients of array s */
```

### See Also

autocoh, crosscoh, crosscorr

## biquad

biquad filter section

### Synopsis

```
#include <filters.h>
float biquad (float sample,
              const float pm coeffs[],
              float dm state[],
              int sections);
```

### Description

The `biquad` function implements a biquad filter. The function produces the filtered response of its input data. The parameter `sections` specifies the number of biquad sections.

The `coeffs` array must be five (5) times the number of sections in length and it also must be located in program memory (PM). The definition is:

```
float pm coeffs[5*sections];
```

The `state` array holds two (2) delay elements per section. It also has one extra location that holds an internal pointer. The total length must be `2*sections + 1`. The definition is:

```
float dm state[2*sections + 1];
```

The state array is not accessed by the user, except that it should be initialized to all zeros before the first call to `biquad`. The first location of `state` is an address. Setting the address to zero tells the function that it is being called for the first time.

### Error Conditions

The `biquad` function does not return an error condition.

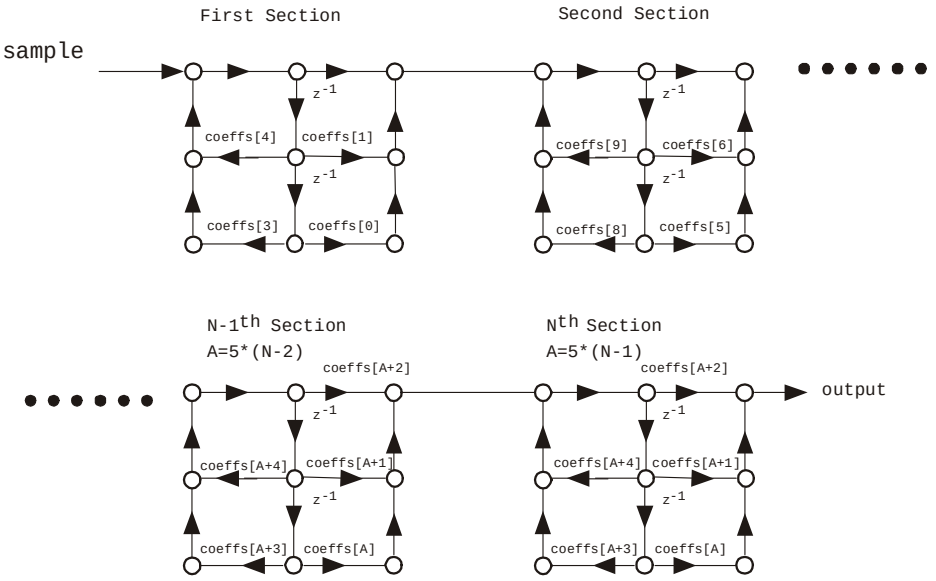
Example

```
#include <filters.h>
#define TAPS 9

float sample, output, state[2*TAPS+1];
float pm coeffs[5*TAPS];
int i;

for (i = 0; i < 2*TAPS+1; i++)
   state[i] = 0;    /* initialize state array */

output = biquad (sample, coeffs, state, TAPS);
```




`N` = the number of biquad sections.

The `coeffs` array holds 5 coefficients for each section and therefore should be `5 * N` in length. The `delay` array holds 2 delayed elements for each section, and should be `2 * N + 1` in length.

The algorithm shown here is adapted from Oppenheim, Alan V. and Ronald Schafer, *Digital Signal Processing*, Englewood Cliffs, New Jersey: Prentice Hall, 1975.

See Also

fir, iir

## cabsf

complex absolute value

### Synopsis

```
#include <complex.h>
float cabsf (complex_float z);
```

### Description

The `cabsf` function returns the floating-point absolute value of its complex input.

The absolute value of a complex number is evaluated with the following formula:

$$y = \sqrt{((Re(x))^2 + (Im(x))^2)}$$

where `x` is the `complex_float` input and `y` is the `float` output.

### Error Conditions

The `cabsf` function does not return an error condition.

### Example

```
#include<complex.h>
complex_float cnum;
float answer;

cnum.re = 12.0;
cnum.im = 5.0;

answer = cabsf (cnum);  /* answer = 13.0 */
```

### See Also

fabs, fabsf, labs

## cexpf

complex exponential

### Synopsis

```
#include <complex.h>
complex_float cexpf (complex_float z);
```

### Description

The `cexpf` function computes the complex exponential value `e` to the power of the first argument.

The exponential of a complex value is evaluated with the following formula:

```
Re(y) = expf (Re(x)) * cosf (Im(x));

Im(y) = expf (Re(x)) * sinf (Im(x));
```

where `x` is the `complex_float` input and `y` is the `complex_float` output.

### Error Conditions

For underflow errors the `cexpf` function returns zero.

### Example

```
#include <complex.h>
complex_float cnum;
complex_float answer;

cnum.re = 1.0;
cnum.im = 0.0;

answer = cexpf (cnum);  /* answer = (2.7182 + 0i) */
```

### See Also

pow, powf, log, logf

## cfftN

N-point complex input fast Fourier transform

### Synopsis

```
#include <trans.h>
float *cfft65536 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft32768 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft16384 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft8192 (const float dm real_input[],
                 const float dm imag_input[],
                 float dm real_output[], float dm imag_output[]);

float *cfft4096 (const float dm real_input[],
                 const float dm imag_input[],
                 float dm real_output[], float dm imag_output[]);

float *cfft2048 (const float dm real_input[],
                 const float dm imag_input[],
                 float dm real_output[], float dm imag_output[]);

float *cfft1024 (const float dm real_input[],
                 const float dm imag_input[],
                 float dm real_output[], float dm imag_output[]);

float *cfft512 (const float dm real_input[],
                const float dm imag_input[],
                float dm real_output[], float dm imag_output[]);

float *cfft256 (const float dm real_input[],
                const float dm imag_input[],
                float dm real_output[], float dm imag_output[]);
```

```
float *cfft128 (const float dm real_input[],
                const float dm imag_input[],
                float dm real_output[], float dm imag_output[]);

float *cfft64 (const float dm real_input[],
               const float dm imag_input[],
               float dm real_output[], float dm imag_output[]);

float *cfft32 (const float dm real_input[],
              const float dm imag_input[],
              float dm real_output[], float dm imag_output[]);

float *cfft16 (const float dm real_input[],
               const float dm imag_input[],
               float dm real_output[], float dm imag_output[]);

float *cfft8 (const float dm real_input[],
             const float dm imag_input[],
             float dm real_output[], float dm imag_output[]);
```

## Description

Each of these 14 `cfftN` functions computes the N-point radix-2 fast Fourier transform (CFFT) of its floating point input (where N is 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536).

There are fourteen distinct functions in this set. They all perform the same function with the same type and number of arguments. The only difference between them is the size of the arrays they operate on. Call a particular function by substituting the number of points for N, as in

```
cfft8 (r_inp, i_inp, r_outp, i_outp);
```

The input to `cfftN` is a floating-point array of N points. If there are fewer than N actual data points, you must pad the array with zeros to make N samples. Better results occur with less zero padding, however. The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data.

`cfftN()` returns a pointer to the `real_output` array.

### Error Conditions

The `cfftN` functions do not return any error conditions.

### Example

```
#include <trans.h>
#define N 2048

float real_input[N], imag_input[N];
float real_output[N], imag_output[N];

/* Real input array is filled from a converter or other source
*/

cfft2048 (real_input, imag_input, real_output, imag_output);
    /* Arrays are filled with FFT data */
```

### See Also

ifftN, rfftN

## copysign, copysignf

copy the sign of the floating-point operand (IEEE arithmetic function)

### Synopsis

```
#include <math.h>
double copysign (double x, double y);
float copysignf (float x, float y);
```

### Description

The `copysign` and `copysignf` functions copy the sign of the second argument `y` to the first argument `x` without changing either its exponent or mantissa. The `copysignf` function is a built-in function which is implemented with an `Fn=Fx COPYSIGN Fy` instruction.

### Error Conditions

This function does not return an error code.

### Example

```
#include <math.h>
double x;
float y;

x = copysign (0.5, -10.0);   /* x = -0.5 */
y = copysignf (-10.0, 0.5f); /* y = 10.0 */
```

### See Also

No references to this function.

The double precision function `copysign` is only available under `-double-size-32` and actually calls the single precision function `copysignf`.

## cot, cotf

cotangent

### Synopsis

```
#include <math.h>
double cot (double x);
float cotf (float x);
```

### Description

The `cot` and `cotf` functions return the cotangent of their argument. The input is interpreted as radians.

The `cot` and `cotf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of $2^{-20}$ over its input range.

### Error Conditions

The `cot` and `cotf` functions do not return an error condition.

### Example

```
#include <math.h>
double x, y;
float v, w;

y = cot (x);
v = cotf (w);
```

### See Also

tan, tanf

## crosscoh

cross-coherence

### Synopsis

```
#include <stats.h>
float *crosscoh (float dm out[],
                 const float dm x[],
                 const float dm y[],
                 int samples,
                 int lags);
```

### Description

The `crosscoh` function computes the cross-coherence of two floating point inputs, `x[]` and `y[]`. The cross-coherence is the cross-correlation minus the product of the mean of `x` and the mean of `y`. The length of the input arrays is given by `samples`. This function returns a pointer to the output data array, `out[]`, of length `lags`.

### Error Conditions

The `crosscoh` function does not return an error condition.

### Example

```
#include <stats.h>
#define SAMPLES 1024

float excitation[SAMPLES], response[16], y[SAMPLES];
int lags = 16;

crosscoh (response, excitation, y, SAMPLES, lags);
```

### See Also

autocoh, autocorr, crosscorr

## crosscorr

cross-correlation

### Synopsis

```
#include <stats.h>
float *crosscorr (float dm out[],
                  const float dm x[],
                  const float dm y[],
                  int samples,
                  int lags);
```

### Description

The `crosscorr` function performs 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 input `x[]` and `y[]` arrays. The length of the input arrays is given by `samples`. This function returns a pointer to the output data array, `out[]`, of length `lags`.

The `crosscorr` function is used in digital signal processing applications such as speech analysis.

### Error Conditions

The `crosscorr` function does not return an error condition.

### Example

```
#include <stats.h>
float r[10], s[160];
float p[160];

crosscorr (r, s, p, 160, 10);
```

### See Also

autocoh, crosscoh, autocorr

## favg, favgf

return mean of two values

### Synopsis

```
#include <math.h>
double favg (double x, double y);
float favgf (float x, float y);
```

### Description

The `favg` and `favgf` functions return the mean of its two arguments. The `favgf` function is a built-in function which is implemented with an `Fn=(Fx+Fy)/2` instruction.

### Error Conditions

The `favg` and `favgf` functions do not return an error code.

### Example

```
#include <math.h>
float x;

x = favgf (10.0f, 8.0f);  /* returns 9.0f */
```

### See Also

avg, lavg

The double-precision function `favg` is only available under `-double-size-32` and actually calls the single precision function `favgf`.

## fclip, fclipf

clip x by y

### Synopsis

```
#include <math.h>
double fclip (double x, double y);
float fclipf (float x, float y);
```

### Description

The `fclip` and `fclipf` functions return the first argument if it is less than the absolute value of the second argument, otherwise it returns the absolute value of the second argument if the first is positive, or minus the absolute value if the first argument is negative. The `fclipf` function is a built-in function which is implemented with an `Fn=CLIP Fx BY Fy` instruction.

### Error Conditions

The `fclip` and `fclipf` functions do not return an error code.

### Example

```
#include <math.h>
float y;

y = fclipf (5.1f, 8.0f);  /* returns 5.1f */
```

### See Also

clip, lclip

The double precision function `fclip` is only available under `-double-size-32` and actually calls the single precision function `fclipf`.

## fir

finite impulse response (FIR) filter

### Synopsis

```
#include <filters.h>
float fir (float sample,
           const float pm coeffs[],
           float dm state[],
           int taps);
```

### Description

The `fir` function implements a finite impulse response (FIR) filter defined by the coefficients and delay line that are supplied in the call of `fir`. The function produces the filtered response of its input data. This FIR filter is structured as a sum of products. The characteristics of the filter (passband, stop band, etc.) are dependent on the coefficient values and the number of taps supplied by the calling program.

The floating-point input to the filter is `sample`. The integer `taps` indicates the length of the filter, which is also the length of the array `coeffs`. The `coeffs` array holds one FIR filter coefficient per element. The coefficients are stored in reverse order; for example, `a_coeffs[0]` holds the `taps -1` (the last coefficient). The `coeffs` array must be located in program memory data space so that the single-cycle dual-memory fetch of the processor can be used.

The `state` array contains a pointer to the delay line as its first element, followed by the delay line values. The length of the `state` array is therefore 1 greater than the number of taps. Each filter has its own `state` array, which should not be modified by the calling program, only by the `fir` function. The `state` array should be initialized to zeros before the `fir` function is called for the first time.

### Error Conditions

The `fir` function does not return an error condition.

### Example

```
#include <filters.h>
float y;
float pm coeffs[10];      /* coeffs array must be */
                          /* initialized and in */
                          /* PM memory */
float state[11];
int i;

for (i = 0; i < 11; i++)
    state[i] = 0;         /* initialize state array */

y = fir (0.775, coeffs, state, 10);
                          /* y holds the filtered output */
```

### See Also

biquad, iir

## fmax, fmaxf

return larger of two values

### Synopsis

```
#include <math.h>
double fmax (double x, double y);
float fmaxf (float x, float y);
```

### Description

The `fmax` and `fmaxf` functions return the larger of its two arguments. The `fmaxf` function is a built-in function which is implemented with an `Fn=MAX(Fx,Fy)` instruction.

### Error Conditions

The `fmax` and `fmaxf` functions do not return an error code.

### Example

```
#include <math.h>
float y;

y = fmaxf (5.1f, 8.0f);   /* returns 8.0f */
```

### See Also

fmin, fminf, lmax, lmin, max, min

The double precision function `fmax` is only available under `-double-size-32` and actually calls the single precision function `fmaxf`.

## fmin, fminf

return smaller of two values

### Synopsis

```
#include <math.h>
double fmin (double x, double y);
float fminf (float x, float y);
```

### Description

The `fmin` and `fminf` functions return the smaller of their two arguments. The `fminf` function is a built-in function which is implemented with an `Fn=MIN(Fx,Fy)` instruction.

### Error Conditions

The `fmin` and `fminf` functions do not return an error code.

### Example

```
#include <math.h>
float y;

y = fminf (5.1f, 8.0f);   /* returns 5.1f */
```

### See Also

fmax, fmaxf, lmax, lmin, max, min

The double precision function `fmin` is only available under `-double-size-32` and actually calls the single precision function `fminf`.

## gen_bartlett

generate bartlett window

### Synopsis

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

### Description

This function generates a vector containing the Bartlett window. The length is specified by parameter `N`. This window is similar to the Triangle window but has the following properties that differ from the Triangle window:

- The Bartlett window always returns a window with two zeros on either end of the sequence. Therefore, for odd `n`, the center section of a `N+2` Bartlett window equals a `N` Triangle window.
- For even `n`, the Bartlett window is 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.

The algorithm used is

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

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

The domain supported by the function is a > 0; N > 0

### Error Conditions

The `gen_bartlett` function does not return an error condition.

### See Also

gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle

## gen_blackman

generate blackman window

### Synopsis

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

### Description

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

The algorithm used is

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

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

The domain supported by the function is a > 0; N > 0

### Error Conditions

The `gen_blackman` function does not return an error condition.

### See Also

gen_bartlett, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle

## gen_gaussian

generate gaussian window

### Synopsis

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

### Description

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

The algorithm used is

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

where n = {0, 1, 2, ..., N-1} and a is an input parameter

The domain supported by the function is a > 0; N > 0; $\boldsymbol{\alpha} > 0.0$

### Error Conditions

The `gen_gaussian` function does not return an error condition.

### See Also

gen_bartlett, gen_blackman, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle

## gen_hamming

generate hamming window

### Synopsis

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

### Description

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

The algorithm used is

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

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

The domain supported by the function is a > 0; N > 0

### Error Conditions

The `gen_hamming` function does not return an error condition.

### See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle

## gen_hanning

generate hanning window

### Synopsis

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

### Description

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

The algorithm used is

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

where n = {1, 1, 2, ..., N+1}

The domain supported by the function is a > 0; N > 0

### Error Conditions

The `gen_hanning` function does not return an error condition.

### See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_harris, gen_kaiser, gen_rectangular, gen_triangle

## gen_harris

generate harris window

### Synopsis

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

### Description

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

The algorithm used is

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

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

The domain supported by the function is a > 0; N > 0

### Error Conditions

The `gen_harris` function does not return an error condition.

### See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_kaiser, gen_rectangular, gen_triangle

## gen_kaiser

generate kaiser window

### Synopsis

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

### Description

This function generates a vector containing the Kaiser window. The length is specified by parameter N. The β value is specified by parameter `b`.

The algorithm used is

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

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

The domain supported by the function is a > 0; N > 0;

### Error Conditions

The `gen_kaiser` function does not return an error condition.

See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_rectangular, gen_triangle

## gen_rectangular

generate rectangular window

### Synopsis

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

### Description

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

The algorithm used is

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

The domain supported by the function is a > 0; N > 0

### Error Conditions

The `gen_rectangular` function does not return an error condition.

### See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_triangle

## gen_triangle

generate triangle window

### Synopsis

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

### Description

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

For even n the following equation applies:

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

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

For odd *n* the following equation applies:

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

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

The domain supported by the function is a > 0; N > 0

## Error Conditions

The `gen_triangle` function does not return an error condition.

## See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular

## histogram

histogram

### Synopsis

```
#include <stats.h>
int *histogram(int dm out[],
               const int dm in[],
               int out_len,
               int samples,
               int bin_size);
```

### Description

The `histogram` function computes a scaled-integer histogram of its input array. The `bin_size` parameter is used to adjust the width of each individual bin in the output array. For example, a `bin_size` of 5 indicates that the first location of the output array holds the number of occurrences of a 0, 1, 2, 3, or 4.

The output array is first zeroed by the function, and then each sample in the input array is multiplied by `1/bin_size` and truncated. The appropriate bin in the output array is incremented. This function returns a pointer to the output array.

All values within the input array must be within range. In order to achieve maximum performance, out of bounds checking is not performed by this function.

### Error Conditions

The `histogram` function does not return an error condition.

## Example

```
#include <stats.h>
#define SAMPLES 1024

int length = 2048;
int excitation[SAMPLES], response[2048];

histogram (response, excitation, length, SAMPLES, 5);
```

## See Also

mean, var

## idle

execute ADSP-21xxx IDLE instruction

### Synopsis

```
#include <21060.h>
void idle (void);
```

### Description

The idle function invokes the ADSP-21xxx `idle` instruction once and returns. The `idle` instruction causes the processor to stop and respond only to interrupts. For a complete description of the `idle` instruction, please refer to the *ADSP-2106x SHARC User's Manual*.

In previous releases of the VisualDSP++ software (prior to release 2.1), the `idle` function repeatedly executed the `idle` instruction. This function has been changed to give you more control over the amount of time spent in the `idle` state.

### Error Conditions

The `idle` function does not return an error condition.

### Example

```
#include <21060.h>

idle ();
```

### See Also

interrupt, interruptf, interrupts, interuptcb, signal

## ifftN

N-point inverse complex input fast Fourier transform (IFFT)

### Synopsis

```
#include <trans.h>
float *ifft65536 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft32768 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft16384 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft8192 (const float dm real_input[],
                 const float dm imag_input[],
                 float dm real_output[], float dm imag_output[]);

float *ifft4096 (const float dm real_input[],
                 const float dm imag_input[],
                 float dm real_output[], float dm imag_output[]);

float *ifft2048 (const float dm real_input[],
                 const float dm imag_input[],
                 float dm real_output[], float dm imag_output[]);

float *ifft1024 (const float dm real_input[],
                 const float dm imag_input[],
                 float dm real_output[], float dm imag_output[]);

float *ifft512 (const float dm real_input[],
                const float dm imag_input[],
                float dm real_output[], float dm imag_output[]);

float *ifft256 (const float dm real_input[],
                const float dm imag_input[],
                float dm real_output[], float dm imag_output[]);
```

```
float *ifft128 (const float dm real_input[],
                const float dm imag_input[],
                float dm real_output[], float dm imag_output[]);

float *ifft64 (const float dm real_input[],
               const float dm imag_input[],
               float dm real_output[], float dm imag_output[]);

float *ifft32 (const float dm real_input[],
               const float dm imag_input[],
               float dm real_output[], float dm imag_output[]);

float *ifft16 (const float dm real_input[],
               const float dm imag_input[],
               float dm real_output[], float dm imag_output[]);

float *ifft8 (const float dm real_input[],
              const float dm imag_input[],
              float dm real_output[], float dm imag_output[]);
```

## Description

Each of these 14 `ifftN` functions computes the N-point radix-2 inverse fast Fourier transform (IFFT) of its floating-point input (where N is 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536).

There are 14 distinct functions in this set. They all perform the same function with the same type and number of arguments. The only difference between them is the size of the arrays on which they operate. To call a particular function, substitute the number of points for N; for example,

```
ifft8 (r_inp, i_inp, r_outp, i_outp);
```

The input to ifftN is a floating-point array of N points. If there are fewer than N actual data points, you must pad the array with zeros to make N samples. Better results occur with less zero padding, however.

The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data. The functions return a pointer to the `real_output` array.

## Error Conditions

The ifftN functions do not return error conditions.

## Example

```
#include <trans.h>
#define N 2048

float real_input[N], imag_input[N];
float real_output[N], imag_output[N];

/* Real input arrays filled from a previous xfft2048() or other
source */

ifft2048 (real_input, imag_input, real_output, imag_output);
    /* Arrays are filled with FFT data */
```

## See Also

cfftN, rfftN

## iir

infinite impulse response (IIR) filter

### Synopsis

```
#include <filters.h>
float iir (float sample,
           const float pm a_coeffs[],
           const float pm b_coeffs[],
           float dm state[],
           int taps);
```

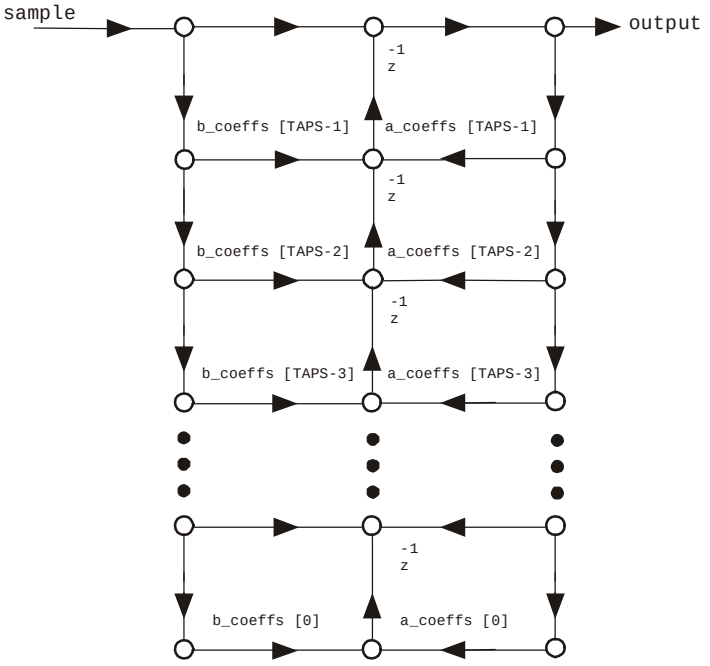
#### Description

The `iir` function implements an infinite impulse response (IIR) filter defined by the coefficients and delay line that are supplied in the call of `iir`. The function produces the filtered response of its input data. The IIR filter implemented in this function is based on the Oppenheim and Schafer Direct Form II. The characteristics of the filter are dependent on the coefficient values supplied by the calling program.

The floating-point input to the filter is `sample`. The integer `taps` indicates the length of the filter, which is the length of the arrays `a_coeffs` and `b_coeffs`. The `a_coeffs` and `b_coeffs` arrays hold one IIR filter coefficient per element. The coefficients are stored in reverse order; so that `a_coeffs[0]` holds the `taps` the last coefficient, and `a_coeffs [taps - 1]` contains the first coefficient. The `coeffs` arrays must be located in program memory data space so that the single-cycle dual-memory fetch of the processor can be used.

The `iir` function stores a pointer to the delay line in the `state` array, in addition to the delay line values. The length of the `state` array is therefore 1 greater than the number of taps. Each filter has its own `state` array, which should not be modified by the calling program, only by the `iir` function. The `state` array should be initialized to zeros before the `iir` function is called for the first time.

The flow graph (below) corresponds to the `iir()` routine as part of the DSP Run-Time Library.




```
The b_coeffs array should equal TAPS+1
The a_coeffs array should equal TAPS
```

## Error Conditions

The `iir` function does not return an error condition.

### Example

```
#include <filters.h>
#define TAPS 10

float pm a_coeffs[TAPS], b_coeffs[TAPS];
float in_sample, output, state[TAPS+1];
int i;

for (i = 0; i < TAPS+1; i++)
    state[i] = 0;   /* initialize state array */

output = iir (in_sample, a_coeffs, b_coeffs, state, TAPS);
```

### See Also

biquad, fir

## matadd

matrix addition

### Synopsis

```
#include <matrix.h>
float *matadd (float output[][],
               const float x_input[][],
               const float y_input[][],
               int r,
               int s);
```

### Description

The `matadd` function performs a matrix addition of the input matrices `x_input[][]` and `y_input[][]`, returning the result in `output[][]`. The `matadd` function returns a pointer to the output matrix. The dimensions of these matrices are `x_input[r][s]`, `y_input[r][s]`, and `output[r][s]`.

### Error Conditions

The `matadd` function does not return an error condition.

### Example

```
#include <matrix.h>

float x[10][20], y[10][20];
float z[10][20];

matadd (z, x, y, 10, 20);
```

### See Also

matsub

## matmul

matrix multiplication

### Synopsis

```
#include <matrix.h>
float *matmul (float z[][],
               const float x[][],
               const float y[][],
               int r,
               int s,
               int t);
```

### Description

The `matmul` function performs a matrix multiplication of the input matrices `x[][]` and `y[][]`, returning a pointer to the `z[][]` output matrix. The dimensions of these matrices are `x[r][s]`, `y[s][t]`, and `z[r][t]`. The matrix multiplication is defined by the following equation: (for i=0 to r-1, for j=0 to t-1):

$$z[i][j] = \sum_{k=0}^{s-1} x[i][k] * y[k][j]$$

### Error Conditions

The `matmul` function does not return an error condition.

### Example

```
#include <matrix.h>

float x[10][5], y[5][10];
float z[10][10];

matmul (z, x, y, 10, 5, 10);
```

See Also

matscalmult

## matscalmult

multiply matrix by scalar

### Synopsis

```
#include <matrix.h>
float *matscalmult (float output[][],
                    const float input[][],
                    float scalar,
                    int r,
                    int s);
```

### Description

The `matscalmult` function performs a scaled multiplication of the `input[][]` matrix, returning the result in `output[][]`. The `input[][]` matrix is multiplied by the input value `scalar`. The `matscalmult` function returns a pointer to the output matrix. The dimensions of these matrices are `input[r][s]`, and `output[r][s]`.

### Error Conditions

The `matscalmult` function does not return an error condition.

### Example

```
#include <matrix.h>
float x[10][5], z[10][5];

matscalmult (z, x, 0.5, 10, 5);
    /* multiplies the matrix x by 0.5 */
```

### See Also

matmul

## matsub

matrix subtraction

### Synopsis

```
#include <matrix.h>
float *matsub (float output[][],
               const float x_input[][],
               const float y_input[][],
               int r,
               int s);
```

### Description

The `matsub` function subtracts the elements of the input matrix `y_input[][]` from the input matrix `x_input[][]`, returning the result in `output[][]`. The `matsub` function returns a pointer to the output matrix. The dimensions of these matrices are `x_input[r][s]`, `y_input[r][s]`, and `output[r][s]`.

### Error Conditions

The `matsub` function does not return an error condition.

### Example

```
#include <matrix.h>

float x[10][5], y[10][5];
float z[10][5];

matsub (z, x, y, 10, 5);
```

### See Also

matadd

## mean

mean of an array of floating point numbers

### Synopsis

```
#include <stats.h>
float mean (const float dm in[], int length);
```

### Description

The `mean` function returns the mean of its floating point input array.

### Error Conditions

The `mean` function does not return an error condition.

### Example

```
#include <stats.h>
float result, input[256];

result = mean (input, 256);
```

### See Also

var

## mu_compress

µ-law compression

### Synopsis

```
#include <comm.h>
int mu_compress (int x);
```

### Description

The `mu_compress` function takes a linear 14-bit signed speech sample and compresses it according to CCITT recommendation G.711. The value returned is an 8-bit sample that can be sent directly to a µ-law codec.

### Error Conditions

The `mu_compress` function does not return an error condition.

### Example

```
#include <comm.h>
int linear, compressed;

compressed = mu_compress (linear);
```

### See Also

mu_expand, a_compress

## mu_expand

μ-law expansion

### Synopsis

```
#include <comm.h>
int mu_expand (int x);
```

### Description

The `mu_expand` function takes an 8-bit compressed speech sample and expands it according to CCITT recommendation G.711 (μ-law definition). The value returned is a linear 14-bit signed sample.

### Error Conditions

The `mu_expand` function does not return an error condition.

### Example

```
#include <comm.h>
int compressed_sample, expanded;

expanded = mu_expand (compressed_sample);
```

### See Also

mu_compress, a_expand

## poll_flag_in

test input flag

### Synopsis

```
#include <21060.h>
int poll_flag_in (int flag, int mode);
```

### Description

The `poll_flag_in` function tests the specified flag (0, 1, 2, 3) for the specified transition (0=low to high, 1=high to low, 2=flag high, 3=flag low, 4=any transition, 5=read flag). The function returns a zero *after* the specified transition has occurred in modes 0-3. In mode 4 it returns the state of the flag after the transition. In mode 5 it returns the value of the flag without waiting.

Table 4-4. poll_flag_in Macros and Values

| Flag Macro | Value | Mode Macro | Value |
|---|---|---|---|
| READ_FLAG0 | 0 | FLAG_IN_LO_TO_HI | 0 |
| READ_FLAG1 | 1 | FLAG_IN_HI_TO_LOW | 1 |
| READ_FLAG2 | 2 | FLAG_IN_HI | 2 |
| READ_FLAG3 | 3 | FLAG_IN_LOW | 3 |
| READ_FLAG3 | 3 | FLAG_IN_TRANSITION | 4 |
| READ_FLAG3 | 3 | RETURN_FLAG_STATE | 5 |

This function assumes that the flag direction in the `MODE2` register is already set as an input (the default state at reset).

### Error Conditions

The `poll_flag_in` function returns a negative value for an invalid flag or transition mode.

### Example

```
#include <21060.h>

poll_flag_in (0, 3);
         /* return zero after transition has occurred */
```

### See Also

interrupt, interruptf, interrupts, interuptcb, set_flag

## rfftN

N-point real input fast Fourier transform

### Synopsis

```
#include <trans.h>
float *rfft65536 (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft32768 (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft16384 (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft8192 (const float dm real_input[],
                 float dm real_output[], float dm imag_output[]);

float *rfft4096 (const float dm real_input[],
                 float dm real_output[], float dm imag_output[]);

float *rfft2048 (const float dm real_input[],
                 float dm real_output[], float dm imag_output[]);

float *rfft1024 (const float dm real_input[],
                 float dm real_output[], float dm imag_output[]);

float *rfft256 (const float dm real_input[],
                float dm real_output[], float dm imag_output[]);

float *rfft128 (const float dm real_input[],
                float dm real_output[], float dm imag_output[]);

float *rfft64 (const float dm real_input[],
               float dm real_output[], float dm imag_output[]);

float *rfft32 (const float dm real_input[],
               float dm real_output[], float dm imag_output[]);
```

```
float *rfft16 (const float dm real_input[],
               float dm real_output[], float dm imag_output[]);

float *rfft8 (const float dm real_input[],
              float dm real_output[], float dm imag_output[]);
```

## Description

Each of these 14 `rfftN` functions are similar to the `cfftN` functions, except that they only take real inputs. They compute the N-point radix-2 fast Fourier transform (RFFT) of their floating point input (where N is 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536).

There are fourteen distinct functions in this set. They all perform the same function with same type and number of arguments. Their only difference is the size of the arrays on which they operate.

Call a particular function by substituting the number of points for N, as in the following example:

```
rfft8 (r_inp,  r_outp,  i_outp);
```

The input to `rfftN` is a floating-point array of N points. If there are fewer than N actual data points, you must pad the array with zeros to make N samples. However, better results occur with less zero padding. The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data.

The `rfftN` functions return a pointer to the `real_output` array.

## Error Conditions

The `rfftN` functions do not return any error conditions.

## Example

```
#include <trans.h>
#define N 2048

float real_input[N];
float real_output[N], imag_output[N];

/* Real input array fills from a converter or other source */

rfft2048 (real_input, real_output, imag_output);
    /* Arrays are filled with FFT data */
```

## See Also

cfftN, ifftN

## rms

root mean square

### Synopsis

```
#include <stats.h>
float rms (const float dm in[], int length);
```

### Description

The `rms` function returns the square root of the mean of the square of its floating-point input array.

### Error Conditions

The `rms` function does not return an error condition.

### Example

```
#include <stats.h>
float input[256], results;

results = rms (input, 256);
```

### See Also

mean, var

## rsqrt, rsqrtf

reciprocal square root

### Synopsis

```
#include <math.h>
double rsqrt (double x);
float rsqrtf (float x);
```

### Description

The `rsqrt` and `rsqrtf` functions return the reciprocal positive square root of their argument.

The `rsqrt` and `rsqrtf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of $2^{-20}$ over its input range.

### Error Conditions

The `rsqrt` and `rsqrtf` functions return zero for a negative input.

### Example

```
#include <math.h>
double y;

y = rsqrt (2.0);    /* y = 0.707 */
```

### See Also

sqrt, sqrtf

## set_flag

set ADSP-21xxx flags

### Synopsis

```
#include <21060.h>
int set_flag (int flag, int mode);
```

### Description

This function is used to set the ADSP-21xxx flags to the desired output value.

The function accepts as input a flag number [0-3] and a mode. The mode can be specified as a macro (defined in `21060.h`) or a value [0-3].

Table 4-5. Flag Function Macros and Values

| Flag Macro | Value | Mode Macro | Value |
|---|---|---|---|
| SET_FLAG0 | 0 | SET_FLAG | 0 |
| SET_FLAG1 | 1 | CLR_FLAG | 1 |
| SET_FLAG2 | 2 | TGL_FLAG | 2 |
| SET_FLAG3 | 3 | TST_FLAG | 3 |

In addition to setting the flag to the specified value, the function also sets the `MODE2` register to specify that the flag is used for output, not input.

If the `TST_FLAG` macro (or a 3) is specified as the mode, the current value (0 or 1) of the flag is returned as the result of the function.

The `set_flag` function returns a zero upon success (except as noted in the previous paragraph).

### Error Conditions

The `set_flag` function returns a non-zero for an error.

### Example

```
#include <21060.h>

set_flag (SET_FLAG0, CLR_FLAG);
set_flag (SET_FLAG0, SET_FLAG);
```

### See Also

poll_flag_in

## set_semaphore

set bus lock semaphore

### Synopsis

```
#include <21060.h>
int set_semaphore (void dm *semaphore, int set_value, int timeout);
```

### Description

The `set_semaphore` function is used to control bus lock in multiprocessor ADSP-21xxx SHARC systems.

A -1 is returned if the bus is locked and the bus lock timeout exceeded.

A 0 is returned if the bus is not locked and a semaphore set.

### Error Conditions

The `set_semaphore` function does not return an error condition.

### See Also

No references to this function.

## timer_off

disable ADSP-21xxx timer

### Synopsis

```
#include <21060.h>
unsigned int timer_off (void);
```

### Description

The `timer_off` function disables the ADSP-21xxx timer and returns the current value of the `TCOUNT` register.

### Error Conditions

The `timer_off` function does not return an error condition.

### Example

```
#include <21060.h>
unsigned int hold_tcount;

hold_tcount = timer_off ();
/* hold_tcount contains value of TCOUNT */
/* register AFTER timer has stopped */
```

### See Also

timer_on, timer_set

The `timer_off` function is not available for the 21065L chip. Refer to timer0_off, timer1_off to disable the ADSP-21065L programmable timers.

The function is supplied only as an in-lined procedure; that is, the compiler substitutes the appropriate statements for any reference to the procedure. Therefore, any source that references `timer_off` must include the `21060.h` header file.

## timer0_off, timer1_off

disable ADSP-21065L timers

### Synopsis

```
#include <21065l.h>
unsigned int timer0_off (void);
unsigned int timer1_off (void);
```

### Description

The `timer0_off` and `timer1_off` functions disable the ADSP-21065L programmable timers and return the current value of the `TCOUNT0` and `TCOUNT1` registers respectively.

### Error Conditions

The `timer0_off` and `timer1_off` functions do not return an error condition.

### Example

```
#include <21065l.h>
unsigned int hold_tcount;

hold_tcount = timer0_off ();
/* hold_tcount contains value of TCOUNT0 */
/* register AFTER timer 0 has stopped */
```

### See Also

timer0_on, timer1_on, timer0_set, timer1_set

The functions are supplied only as in-lined procedures; that is, the compiler substitutes the appropriate statements for any reference to the procedures. Therefore, any source that references either `timer0_off` or t`imer1_off` must include the `21065l.h` header file.

## timer_on

enable ADSP-21xxx timer

### Synopsis

```
#include <21060.h>
unsigned int timer_on (void);
```

### Description

The `timer_on` function enables the ADSP-21xxx timer and returns the current value of the `TCOUNT` register.

### Error Conditions

The `timer_on` function does not return an error condition.

### Example

```
#include <21060.h>
unsigned int hold_tcount;

hold_tcount = timer_on ();
 /* hold_tcount contains value of TCOUNT */
 /* register when timer starts */
```

### See Also

timer_off, timer_set

The `timer_on` function is not available for the 21065L chip. Refer to timer0_on, timer1_on to enable the ADSP-21065L programmable timers.

The function is supplied only as an in-lined procedure; that is, the compiler substitutes the appropriate statements for any reference to the procedure. Therefore, any source that references `timer_on` must include the `21060.h` header file.

## timer0_on, timer1_on

enable ADSP-21065L timers

### Synopsis

```
#include <21065l.h>
unsigned int timer0_on (void);
unsigned int timer1_on (void);
```

### Description

The `timer0_on` and `timer1_on` functions enable the ADSP-21065L programmable timers and return the current value of the `TCOUNT0` and `TCOUNT1` registers respectively.

### Error Conditions

The `timer0_on` and `timer1_on` functions do not return an error condition.

### Example

```
#include <21065l.h>
unsigned int hold_tcount;

hold_tcount = timer0_on ();
/* hold_tcount contains value of TCOUNT0 */
/* register when timer 0 starts */
```

### See Also

timer0_off, timer1_off, timer0_set, timer1_set

The functions are supplied only as in-lined procedures; that is, the compiler substitutes the appropriate statements for any reference to the procedures. Therefore, any source that references either `timer0_on` or `timer1_on` must include the `21065l.h` header file.

## timer_set

initialize ADSP-21xxx timer

### Synopsis

```
#include <21060.h>
int timer_set (unsigned int tperiod,
               unsigned int tcount);
```

### Description

The `timer_set` function sets the ADSP-21xxx timer registers `TPERIOD` and `TCOUNT`. The function returns a 1 if the timer is enabled, or a zero if the timer is disabled.

Each interrupt call takes approximately 50 cycles on entrance and 50 cycles on return. If `TPERIOD` and `TCOUNT` are set too low, you may incur an initializing overhead that could create an infinite loop.

### Error Conditions

The `timer_set` function does not return an error condition.

### Example

```
#include <21060.h>

if (timer_set (1000, 1000) != 1)
    timer_on ();    /* enable timer */
```

### See Also

timer_on, timer_off

The `timer_set` function is not available for the 21065L chip. Refer to timer0_set, timer1_set to initialize the ADSP-21065L programmable timers.

The function is supplied only as an in-lined procedure; that is, the compiler substitutes the appropriate statements for any reference to the procedure. Therefore, any source that references `timer_set` must include the `21060.h` header file.

## timer0_set, timer1_set

initialize ADSP-21065L timers

### Synopsis

```
#include <21065l.h>
int timer0_set (unsigned int tperiod,
                unsigned int tcount,
                unsigned int tscale);
int timer1_set (unsigned int tperiod,
                unsigned int tcount,
                unsigned int tscale);
```

### Description

The `timer0_set` and `timer1_set` functions set the ADSP-21065L timer registers `TPERIOD0`, `TCOUNT0`, `TPWIDTH0` and `TPERIOD1`, `TCOUNT1`, `TPWIDTH1` respectively. The functions return a 1 if the corresponding timer is enabled, or a zero if the timer is disabled.

Each interrupt call takes approximately 50 cycles on entrance and 50 cycles on return. If `TPERIOD` and `TCOUNT` are set too low, you may incur an initializing overhead that could create an infinite loop.

### Error Conditions

The `timer0_set` and `timer1_set` functions do not return an error condition.

### Example

```
#include <21065l.h>
unsigned int hold_tcount;

if (timer0_set (200, 1, 150) != 1)
    timer0_on ();    /* enable timer 0 */
```

See Also

timer0_off, timer1_off, timer0_on, timer1_on

The functions are supplied only as in-lined procedures; that is, the compiler substitutes the appropriate statements for any reference to the procedures. Therefore, any source that references either `timer0_set` or `timer1_set` must include the `21065l.h` header file.

## var

variance

### Synopsis

```
#include <stats.h>
float var (const float dm in[], int length);
```

### Description

The `var` function returns the variance of its floating-point input array.

### Error Conditions

The `var` function does not return an error condition.

### Example

```
#include <stats.h>
float input[256], result;

result = var (input, 256);
```

### See Also

mean

## zero_cross

count zero crossings

### Synopsis

```
#include <stats.h>
int zero_cross (const float dm in[], int length);
```

### Description

The `zero_cross` function returns the number of times that a signal represented in the input array crosses over the zero line. If all input values are zero, the function returns a zero.

### Error Conditions

The `zero_cross` function does not return an error condition.

### Example

```
#include <stats.h>
float input[256];
int results;

results = zero_cross (input, 256);
```

### See Also

No references to this function.