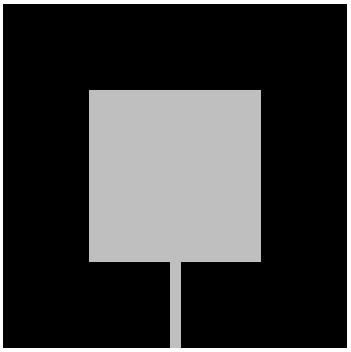
ADSP-2100 Family

C Runtime Library Manual



a

ADSP-2100 Family
C Runtime Library Manual

PRINTED IN U.S.A.

Printing History
SECOND EDITION 11/94

For marketing information or Applications Engineering assistance, contact your local Analog Devices sales office or authorized distributor.

If you have suggestions for how the ADSP-2100 Family development tools or documentation can better serve your needs, or you need Applications Engineering assistance from Analog Devices, please contact:

Analog Devices, Inc.
DSP Applications Engineering
One Technology Way
Norwood, MA 02062-9106
Tel: (617) 461-3672
Fax: (617) 461-3010
e-mail: dsp_applications@analog.com

Or log in to the DSP Bulletin Board System:
Telephone number (617) 461-4258
300, 1200, 2400, 9600 baud, no parity, 8 bits data, 1 stop bit

For additional marketing information, call (617) 461-3881 in Norwood MA, USA.

Literature

ADSP-2100 FAMILY MANUALS

ADSP-2100 Family User's Manual (Prentice Hall)
Complete description of processor architectures and system interfaces.

ADSP-2100 Family Assembler Tools & Simulator Manual
ADSP-2100 Family C Tools Manual
ADSP-2100 Family C Runtime Library Manual
Programmer's references.

ADSP-2100 Family EZ Tools Manual
User's manuals for in-circuit emulators and demonstration boards.

APPLICATIONS INFORMATION

Digital Signal Processing Applications Using the ADSP-2100 Family, Volume 1 (Prentice Hall)
Topics include arithmetic, filters, FFTs, linear predictive coding, modem algorithms, graphics, pulse-code modulation, multirate filters, DTMF, multiprocessing, host interface and sonar.

Digital Signal Processing Applications Using the ADSP-2100 Family, Volume 2 (Prentice Hall)
Topics include modems, linear predictive coding, GSM codec, sub-band ADPCM, speech recognition, discrete cosine transform, digital tone detection, digital control system design, IIR biquad filters, software uart and hardware interfacing.

SPECIFICATION INFORMATION

ADSP-2100/ADSP2100A Data Sheet
ADSP-21xx Data Sheet
ADSP-21msp50A/55A/56A Data Sheet
ADSP-21msp58/59 Preliminary Data Sheet
ADSP-2171/72/73 Data Sheet
ADSP-2181 Preliminary Data Sheet

Contents

Contents

LIBRARY FUNCTIONS & DESCRIPTIONS

Contents

TABLES

Contents

Introduction 1

1.1 OVERVIEW

The C Runtime Library Manual documents a set of C callable functions written in ADSP-2100 Family assembly language. Programs written in C depend on library functions to perform basic operations not provided by the language. These functions include memory allocation, signal processing, and mathematics functions. Use of the library simplifies the software development process.

This document describes the current release of the runtime library. Future releases may include more functions.

The object files of the library functions may be freely used in systems based on ADSP-2100 Family processors.

The algorithms used to implement many of the mathematic functions are taken from the following reference:

> Cody and Waite. *Software Manual For The Elementary Functions.* Prentice Hall, 1980.

See the current *ADSP-2100 Family Development Software Tools* release note for function benchmarks.

1.2 DOCUMENTATION

1.2.1 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.

1 Introduction

1.2.2 Reference Format

Each function in the library is listed in the following format:

Description	***Explanation***
FUNCTION foof—does nothing	*Purpose of the function*

SYNOPSIS
`#include <fnord.h>`
`void foof(float x);`
	Required header file and functional prototype

DESCRIPTION The `foof` function...	*Function specification*
ERROR CONDITIONS Errors returned by the `foof`...	*How the function indicates an error*

EXAMPLE
`#include <fnord.h>`
`foof(0.0);`
	Typical function usage

SEE ALSO `foo2, foo3`	*Related functions*

Using Library Functions 2

2.1 CALLING FUNCTIONS

To use a C library function, call the function by name and give the appropriate arguments. The necessary arguments for each function are specified in the *Library Reference* section of this volume.

You can use the librarian, `lib21`, documented in the *ADSP-2100 Family Assembler Manual*, to build libraries of your own functions.

2.1.1 Header Files

If your program calls a library function, you must include the header file containing the function with the `#include` C preprocessor directive. For the header file to include, see the *Synopsis* section of the library function you call.

Header files contain function prototypes. The compiler uses these prototypes to check that your program calls each function with the correct arguments.

2.1.2 Built-In Functions

The C Runtime Library built-in functions are a small set of functions that the compiler immediately recognizes and replaces with inline assembly code instead of a function call. Typically, inline assembly code is faster than an average library routine and it does not incur calling overhead. For example, `strcpy()`, a function used to copy one string into another space, is recognized by the compiler, which subsequently replaces a call to the C Runtime Library version with an inline version.

The following routines are built-in functions for the G21 compiler:

```
abs()           strcmp()
fabs()      strcpy()
labs()      strlen()
memcpy()    sqrt()
memcmp()
```

If you want to use the C Runtime Library functions of the same name, compile with the `-fno-builtin` G21 command line switch.

2 Using Library Functions

2.2 LIBRARY DIRECTORY

The linker looks for the C Runtime Library, `libda.a`, in the sub-directory `\21xx\lib` of the directory specified in the `ADI_DSP` operating system environment variable. For example, in an MS-DOS system with the value of `ADI_DSP` set to `c:\adi_dsp`, the linker looks for the C Runtime Library in the directory `c:\adi_dsp\21xx\lib`.

2.3 LIBRARY SOURCE CODE

The distribution disks contain source code for the functions and macros in the C Runtime Library. By default, the installation program copies the source code to a subdirectory of the directory where the runtime libraries are kept, named `\21xx\lib\src`. For example, in an MS-DOS system, the source code for the library is found in the `c:\adi_dsp\21xx\lib\src` directory. Each function is kept in a separate file. If you do not intend to modify any of the C Runtime 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. Customizing the library requires knowledge of both ADSP-2100 Family assembly language and the C Runtime environment. For information on ADSP-2100 Family assembly language, refer to the *ADSP-2100 Family User's Manual*. For information on the C Runtime environment, see the *ADSP-2100 Family C Tools Manual*. Copy the source code to a file with a different filename before you make any modifications to the source code. Use a new, unique name for your modified function.

Note: Analog Devices only supports the runtime library functions as provided.

Library Structure 3

The C Runtime Library has several categories of functions and macros defined by the ANSI C standard, plus extensions provided by Analog Devices.

3.1 ANSI RUNTIME ENVIRONMENT MACROS & DEFINITIONS

The error handling, standard definitions, limits, and floating-point categories do not contain individual functions; they consist of macros and type definitions.

Header	*Purpose*
`errno.h`	Error Handling
`stddef.h`	Standard Definitions
`limits.h`	Limits
`float.h`	Floating Point

Table 3.1 ANSI Standard Runtime Environment Macros & Definitions

3.2 ANSI STANDARD FUNCTIONS

Header	*Purpose*
`math.h`	Mathematics
`signal.h`	Signal Handling
`stdarg.h`	Variable Arguments
`ctype.h`	Character Handling
`string.h`	String Handling
`stdlib.h`	Standard Library
`locale.h`	Localization
`assert.h`	Diagnostics
`setjump.h`	Non-Local Jumps
`stdio.h`	Input/Output

Table 3.2 ANSI Standard Functions

Note: The `stdio.h` file only contains the definition for `EOF`. The C Runtime Library does not provide standard I/O routines.

3 Library Structure

FUNCTION
abs—absolute value

SYNOPSIS
```
#include <stdlib.h>
int abs(int j);
```

DESCRIPTION
The `abs` function returns the absolute value of its `int` input.

Note: `abs(INT_MIN)` returns `INT_MIN`.

ERROR CONDITIONS
The `abs` function does not return an error condition.

EXAMPLE
```
#include <stdlib.h>
int i;

i = abs(-5); /* i == 5 */
```

SEE ALSO
`fabs, labs`

FUNCTION

acos—arc cosine

SYNOPSIS

```
#include <math.h>
double acos(double);
```

DESCRIPTION

The `acos` function returns the arc cosine of `x`. The input must be in the range [-1, 1]. The output, in radians, is in the range [0, π].

The `acos` function returns 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 `acos` function indicates a domain error (by setting `errno` to `EDOM`) and returns a zero if the input is not in the range [-1, 1].

EXAMPLE

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

y = acos(0.0);    /* y = π/2 */
```

SEE ALSO

`cos`

FUNCTION
asin—arc sine

SYNOPSIS
```
#include <math.h>
double asin(double);
```

DESCRIPTION
The `asin` function returns the arc sine of the argument. The input must be in the range [-1, 1]. The output, in radians, is in the range $[-\pi/2, \pi/2]$.

The `asin` function returns 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 `asin` function indicates a domain error (set `errno` to `EDOM`) and returns a zero if the input is not in the range [-1, 1].

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

y = asin(1.0);    /* y = π/2 */
```

SEE ALSO
`sin`

FUNCTION

atan—arc tangent

SYNOPSIS

```
#include <math.h>
double atan(double);
```

DESCRIPTION

The `atan` function returns the arc tangent of the argument. The output, in radians, is in the range $[-\pi/2, \pi/2]$.

The `atan` function returns 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 `atan` function does not return an error condition.

EXAMPLE

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

y = atan(0.0);    /* y = 0.0 */
```

SEE ALSO

atan2, tan

FUNCTION
atan2—arc tangent of quotient

SYNOPSIS
```
#include <math.h>
double atan2(double x, double y);
```

DESCRIPTION
The `atan2` function computes the arc tangent of the input value `x` divided by input value `y`. The output, in radians, is in the range $[-\pi, \pi]$.

The `atan2` function returns 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 `atan2` function returns a zero and sets `errno` to `EDOM` if x=0 and y <> 0.

EXAMPLE
```
#include <math.h>
double A;

A = atan2(0.0/0.5);    /* A = 0.0 */
```

SEE ALSO
`atan, tan`

FUNCTION
ceil—ceiling

SYNOPSIS
```
#include <math.h>
double ceil(double);
```

DESCRIPTION
The `ceil` function returns the smallest integral value, expressed as double, that is not less than its input.

ERROR CONDITIONS
The `ceil` function does not return an error condition.

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

y = ceil(1.05);   /* y = 2.0 */
```

SEE ALSO
`floor`

FUNCTION
cos—cosine

SYNOPSIS
```
#include <math.h>
double cos(double);
```

DESCRIPTION
The `cos` function returns the cosine of the argument. The input is interpreted as radians; the output is in the range [-1, 1].

The `cos` function returns 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.

Although the `cos` function accepts input over the entire floating-point range, the accuracy of the result decreases significantly for a large input (greater than $\pi^{12}/2$).

ERROR CONDITIONS
The `cos` function does not return an error condition.

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

y = cos(3.14159);      /* y = -1.0 */
```

SEE ALSO
acos

FUNCTION
cosh—hyperbolic cosine

SYNOPSIS
```
#include <math.h>
double cosh(double);
```

DESCRIPTION
The `cosh` function returns the hyperbolic cosine of its argument and 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 `cosh` function returns `HUGE_VAL` and sets `errno` to `ERANGE` if the input exceeds 2^{12}.

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

y = cosh(x);
```

SEE ALSO
`sinh`

remove the mean of a data buffer — demean_buffer

FUNCTION
demean_buffer—remove the mean of a data buffer

SYNOPSIS
```
#include <filters.h>
int demean_buffer (int *input_buffer, int old_mean,
          int LENGTH);
```

DESCRIPTION
The `demean_buffer()` routine removes a DC-bias from input signals and the mean from a buffer of data. It can also execute a notch filter on the input based on an adaptive filter. (See *Adaptive Signal Processing*, Prentice Hall, 1985.)

`demean_buffer` returns the mean of the current buffer as a result. This value should be passed as a parameter to the function on the next call; the first call to `demean_buffer` should have a 0 for the `old_mean` value.

ERROR CONDITIONS
The `demean_buffer` function does not return error conditions.

EXAMPLE
```
#define BUFSIZE 1024

int data_buffer [BUFSIZE];
int data_mean = 0;

/* The buffer is filled with data, possibly from a
          */
/* converter. Remove the mean from the buffer with
          */
/* this demean function:                          */

     data_demean = demean_buffer(data buffer,
```

```
           data_mean, BUFSIZE);

  /* Or, like this example: */

       {
       int i, temp_mean;

       temp_mean = 0;
       for (i=0; i<BUFSIZE; i++)
        temp_mean += data_buffer[i];

       temp_mean /= BUFSIZE;

       for(i=0; i<BUFSIZE; i++)
        data_buffer[i] -= temp_mean;

       }
```

SEE ALSO

No references to this function.

FUNCTION
exp—exponential

SYNOPSIS
```
#include <math.h>
double exp(double);
```

DESCRIPTION
The `exp` function computes the exponential value `e` to the power of its argument.

ERROR CONDITIONS
The `exp` function returns the value `HUGE_VAL` and stores the value `ERANGE` in `errno` when there is an overflow error. In the case of underflow, the `exp` function returns a zero.

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

y = exp(1.0);     /* y = 2.71828...*/
```

SEE ALSO
`pow, log`

FUNCTION
fabs—float absolute value

SYNOPSIS
```
#include <math.h>
double fabs(double);
```

DESCRIPTION
The `fabs` function returns the absolute value of the argument.

ERROR CONDITIONS
The `fabs` function does not return an error condition.

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

y = fabs(-2.3);   /* y = 2.3 */
y = fabs(2.3);    /* y = 2.3 */
```

SEE ALSO
abs, labs

N-point complex input fast Fourier transform (FFT)

fftN

FUNCTION

fftN—N-point complex input fast Fourier transform (FFT)

SYNOPSIS

```
int fft1024(int rl_in[],
     int im_in[],
     int rl_out[],
     int im_out[]);

int fft512(int rl_in[],
     int im_in[],
     int rl_out[],
     int im_out[]);

int fft256(int rl_in[],
     int im_in[],
     int rl_out[],
     int im_out[]);

int fft128(int rl_in[],
     int im_in[],
     int rl_out[],
     int im_out[]);

int fft64(int rl_in[],
     int im_in[],
     int rl_out[],
     int im_out[]);

int fft32(int rl_in[],
     int im_in[],
     int rl_out[],
     int im_out[]);

int fft16(int rl_in[],
     int im_in[],
     int rl_out[],
     int im_out[]);

int fft8(int rl_in[],
     int im_in[],
     int rl_out[],
     int im_out[]);
```

DESCRIPTION

These functions are Analog Devices extensions to the ANSI standard.

Each of these 8 `fftN` functions computes the N-point radix-2 fast Fourier transform (FFT) of its integer input (where N is 8, 16, 32, 64, 128, 256, 512, or 1024).

There are 8 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,

```
fft8(r_inp, i_inp, r_outp, i_outp);
```

not

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

The input to `fftN` is a integer 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 block exponent.

ERROR CONDITIONS

The `fftN` functions do not return error conditions.

EXAMPLE

```
#include <fft.h>
#define  N 1024

int i;
int real_input[N], imag_input[N], imag_output[N];
int real_output[N];


fft1024 (real_input, imag_input, real_output,
          imag_output) ;
     /* Arrays are filled with FFT data */
```

SEE ALSO

`ifftN`

FUNCTION
fir—finite impulse response (FIR) filter

SYNOPSIS
```
#include <filters.h>

int fir(int sample, int pm coeffs[], \
     int dm state[], int taps);
```

DESCRIPTION
This function is an Analog Devices extension to the ANSI standard.

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 integer 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, `coeffs[0]` holds the `(taps-1)` coefficient. The `coeffs` array is located in program memory data space to use the single-cycle dual-memory fetch of the processor.

The `state` array contains a pointer to the delay line as its last element, preceeded 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.

The parameters `sample`, `coeffs[]`, and `state[]`, are all considered to be fractional numbers. The `fir()` executes fractional multiples that preserve the format of the fractional input. If your application requires a true integer `fir()`, you should divide the ouput of the filter by two.

ERROR CONDITIONS

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

EXAMPLE

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

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

y = fir(0x1234,coeffs, state, 10);
                  /* y holds the filtered output */
```

SEE ALSO

iir

FUNCTION
floor—floor

SYNOPSIS
```
#include <math.h>
double floor(double);
```

DESCRIPTION
The `floor` function produces the largest integral value that is not greater than its input.

ERROR CONDITIONS
The `floor` function does not return an error condition.

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

y = floor(1.25);       /* y = 1.0 */
y = floor(-1.25);      /* y = -2.0 */
```

SEE ALSO
`ceil`

FUNCTION

fmod—floating-point modulus

SYNOPSIS

```
#include <math.h>
double fmod(double, double);
```

DESCRIPTION

The `fmod` function computes the floating-point remainder that results from dividing the first argument into the second argument.

This value is less than the second argument and has the same sign as the first argument. If the second argument is equal to zero, `fmod` returns a zero.

ERROR CONDITIONS

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

EXAMPLE

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

y = fmod(5.0, 2.0);          /* y = 1.0 */
```

SEE ALSO

modf

FUNCTION
frexp—separate fraction and exponent

SYNOPSIS
```
#include <math.h>
double frexp(double, int*);
```

DESCRIPTION
The `frexp` function separates a floating-point input into a normalized fraction and a (base 2) exponent.

The function returns the first argument, a fraction in the interval [½, 1), and stores a power of 2 in the integer pointed to by the second argument. If the input is zero, then zeros are stored in both arguments.

ERROR CONDITIONS
The `frexp` function does not return an error condition.

EXAMPLE
```
#include <math.h>
double y;
int exponent;

y = frexp(2.0, &exponent);  /* y=0.5, exponent=2 */
```

SEE ALSO
modf

FUNCTION

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

SYNOPSIS

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

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

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

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

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

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

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

int *ifft512(int dm real_input[],
     int dm imag_input[],
     int dm real_output[],
     int dm imag_output[]);
```

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

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

int *ifft128(int dm real_input[],
     int dm imag_input[],
     int dm real_output[],
     int dm imag_output[]);

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


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

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

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

DESCRIPTION

These functions are Analog Devices extensions to the ANSI standard.

Each of these 14 `ifftN` functions computes the N-point radix-2 inverse fast Fourier transform (IFFT) of its integer 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);
```

not

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

The input to `ifftN` is a integer 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 <fft.h>
#define  N 2048

int i;
int real_input[N], imag_input[N], imag_output[N];
int real_output[N];

     /* Real input array is filled from a previous
          fft2048() or other source*/

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

SEE ALSO

`fftN`

FUNCTION

iir—infinite impulse response (IIR) filter

SYNOPSIS

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

DESCRIPTION

This function is an Analog Devices extension to the ANSI standard.

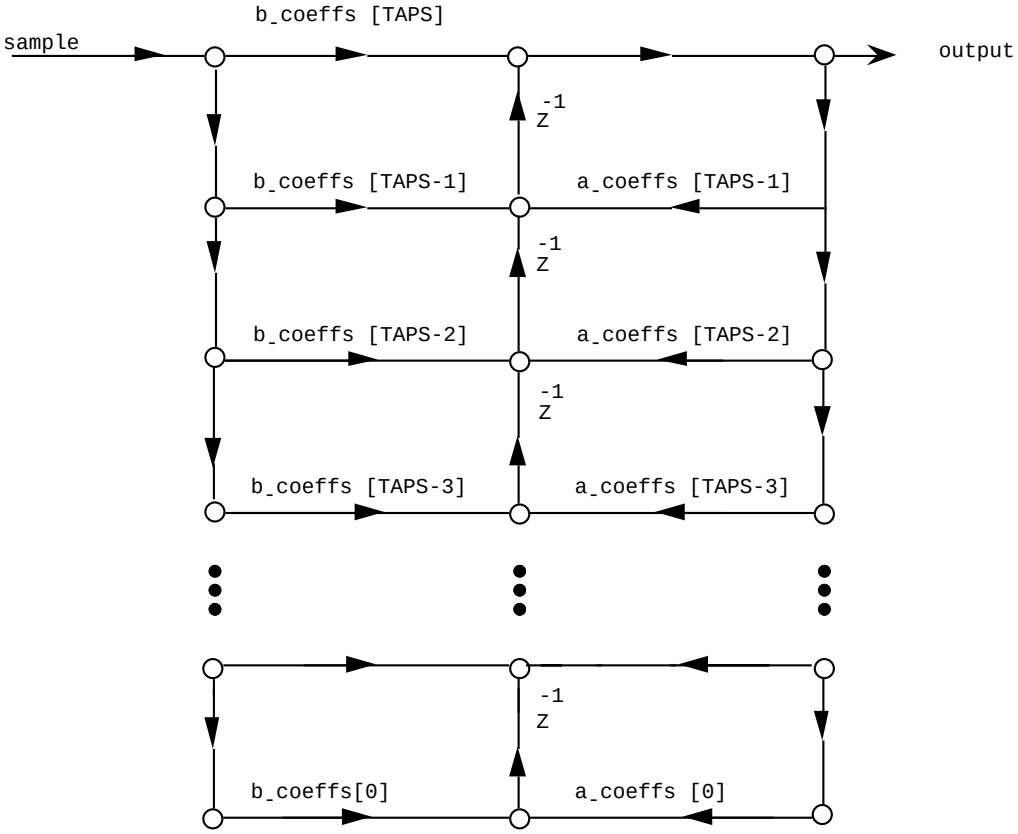
The `iir` function implements an infinite impulse response (IIR) filter defined by the coefficients and delay line that are supplied in the call of `iir`. The function produces the filtered response of its input data. The IIR filter implemented in this function is based on the Oppenheim and Schafer Direct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

The integer input to the filter is `sample`. The integer `taps` indicates the length of the filter, which is the length of the arrays `a_coeffs`, `b_coeffs` and `delay`. The `a_coeffs` and `b_coeffs` arrays hold one IIR filter coefficient per element. The coefficients are stored in reverse order; for example, `a_coeffs[0]` holds the `taps-1` (the last coefficient).

The `state` array contains a pointer to the delay line as its last element, preceeded 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 `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 C Runtime Library*:

iir ( )




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

* Adapted from Oppenheim and Schafer,
Digital Signal Processing, New Jersey: Prentice Hall, 1975.

ERROR CONDITIONS

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

EXAMPLE

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

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

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

`fir`

FUNCTION
interrupt—define interrupt handling

SYNOPSIS
```
#include <signal.h>
void (*interrupt (int sig, void(*func)(int))) (int);
```

DESCRIPTION
This function is an Analog Devices extension to the ANSI standard.

The `interrupt` function determines how a signal received during program execution is handled. The `interrupt` function executes the function pointed to by `func` at every interrupt; the `signal` function executes the function only once. The `sig` argument must be one of the following that are listed in priority order below:

Signal Value	*Meaning*	
SIG_IGN	Signal Ignore	
SIG_DFL	Signal Default	
SIG_ERR	Signal Error	
SIGINT3	Interrupt 3	<ADSP-2100 Only>
SIGINT2	Interrupt 2	
SIGINT1	Interrupt 1	
SIGINT0	Interrupt 0	
SIGSPORT0XMIT	Signal Sport 0 Transmit	<Not ADSP-2100>
SIGSPORT0RECV	Signal Sport 0 Receive	
SIGTIMER	Signal Timer	
SIGSPORT1XMIT	Signal Sport 1 Transmit	<Not ADSP-2100>
SIGSPORT1RECV	Signal Sport 1 Receive	<Not ADSP-2100>
SIGHIPWRITE	Signal HIP Write	<ADSP-2111, ADSP-2150, ADSP-2171 Only>
SIGHIPREAD	Signal HIP Read	<ADSP-2111, ADSP-2150, ADSP-2171 Only>
SIGPOWERDOWN	Signal Power Down	<ADSP-2150, ADSP-2171>
SIGCODECXMIT	Signal Codec Transmit	<ADSP-2150 Only>
SIGCODECREC	Signal Codec Record	<ADSP-2150 Only>
SIGSWI1	Signal Software Interupt 1	<ADSP-2171 Only >
SIGSWI0	Signal Software Interupt 0	<ADSP-2171 Only >

The `interrupt` function causes the receipt of the signal number `func` to be handled in one of the following ways:

`func` value	*action*
`SIG_DFL`	The default action is taken.
`SIG_IGN`	The signal is ignored.
`Function Address`	The function pointed to by `func` is executed.

The function pointed to by `func` is executed each time the interrupt is received. The `interrupt` function must be called with the `SIG_IGN` argument to disable interrupt handling.

Note: Interrupts are nested by default.

ERROR CONDITIONS

The `interrupt` function returns `SIG_ERR` and sets `errno` equal to `SIG_ERR` if the requested interrupt is not recognized.

EXAMPLE

```
#include <signal.h>

interrupt(SIGINT2, int2_handler);
     /* enable interrupt 2 whose handling routine is
          pointed to by int2_handler */

interrupt(SIGINT2, SIG_IGN);
     /* disable interrupt 2 */
```

SEE ALSO

`signal, raise`

FUNCTION

isalpha—detect alphabetic character

SYNOPSIS

```
#include <ctype.h>
int isalpha(int c);
```

DESCRIPTION

The `isalpha` function determines if the input is an alphabetic character (`A-Z` or `a-z`). If the input is not alphabetic, `isalpha` returns a zero. If the input is alphabetic, `isalpha` returns a nonzero value.

ERROR CONDITIONS

The `isalpha` function does not return any error conditions.

EXAMPLE

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
     printf("%#04x", ch);
     printf("%2s", isalpha(ch) ? "alphabetic" : "");
     putchar('\n');
     }
```

SEE ALSO

isdigit

FUNCTION
isdigit—detect decimal digit

SYNOPSIS
```
#include <ctype.h>
int isdigit(int c);
```

DESCRIPTION
The `isdigit` function determines if the input character is a decimal digit (0-9). If the input is not a digit, `isdigit` returns a zero. If the input is a digit, `isdigit` returns a nonzero value.

ERROR CONDITIONS
The `isdigit` function does not return an error condition.

EXAMPLE
```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
     printf("%#04x", ch);
     printf("%2s", isdigit(ch) ? "digit" : "");
     putchar('\n');
     }
```

SEE ALSO
isalpha

FUNCTION
labs—long integer absolute value

SYNOPSIS
```
#include <stdlib.h>
long int labs(long int j);
```

DESCRIPTION
The `labs` function returns the absolute value of its integer input.

ERROR CONDITIONS
The `labs` function does not return an error condition.

EXAMPLE
```
#include <stdlib.h>
long int j;

j= labs(-285128); /* j = 285128 */
```

SEE ALSO
`abs, fabs`

FUNCTION

ldexp—multiply by power of 2

SYNOPSIS

```
#include <math.h>
double ldexp(double, int);
```

DESCRIPTION

The `ldexp` function returns the value of the floating-point input multiplied by 2^n. It adds the value of `n` to the exponent of `x`.

ERROR CONDITIONS

If the result overflows, `ldexp` returns `HUGE_VAL` with the proper sign and sets `errno` to `ERANGE`. If the result underflows, a zero is returned.

EXAMPLE

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

y = ldexp(0.5, 2);          /* y = 2.0 */
```

SEE ALSO

exp, pow

FUNCTION

log—natural logarithm

SYNOPSIS

```
#include <math.h>
double log(double);
```

DESCRIPTION

The `log` function produces the appropriate precision natural (base e) logarithm of its input.

The `log` function returns 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 `log` function returns a zero and sets `errno` to `EDOM` if the input value is negative.

EXAMPLE

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

y = log(1.0);          /* y = 0.0 */
```

SEE ALSO

`exp, log, log10`

FUNCTION
log10—base 10 logarithm

SYNOPSIS
```
#include <math.h>
double log10(double);
```

DESCRIPTION
The `log10` function produces the base 10 logarithm of its input and returns 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 `log10` function indicates a domain error (set `errno` to `EDOM`) and returns a zero if the input is negative.

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

y = log10(100.0);      /* y = 2.0 */
```

SEE ALSO
`log, pow`

FUNCTION

memcmp—compare objects

SYNOPSIS

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t
          n);
```

DESCRIPTION

The `memcmp` function compares the first `n` characters of the objects pointed to by `s1` and `s2`. It returns a positive lexical value if the `s1` object is greater than the `s2` object. If the `s2` object is greater than the `s1` object, a negative value is returned. A zero is returned if the objects are the same.

ERROR CONDITIONS

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

EXAMPLE

```
#include <string.h>
char string1 = “ABC”;
char string2 = “BCD”;

result=memcmp (string1, string2, 3);  /* result<0 */
```

SEE ALSO

`strcmp, strncmp`

copy characters from one object to another — memcpy

FUNCTION
memcpy—copy characters from one object to another

SYNOPSIS
```
#include <string.h>
void *memcpy(void *s1, const void *s2,
     size_t n);
```

DESCRIPTION
The `memcpy` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The behavior of `memcpy` is undefined if the two objects overlap.

The `memcpy` function returns the address of `s1`.

ERROR CONDITIONS
The `memcpy` function does not return an error condition.

EXAMPLE
```
#include <string.h>
char *a = “SRC”;
char *b = “DEST”;

result=memcpy (b, a, 3);    /* *b=”SRC” */
```

SEE ALSO
strcpy, strncpy

FUNCTION
memset—set range of memory to a character

SYNOPSIS
```
#include <string.h>
void *memset(void *s1, int c, size_t n);
```

DESCRIPTION
The `memset` function sets a range of memory to the input character `c`. The first `n` characters of `s1` are set to `c`.

The `memset` function returns a pointer to `s1`.

ERROR CONDITIONS
The `memset` function does not return an error condition.

EXAMPLE
```
#include <string.h>
char string1[50];

memset(string1, '\0', 50);  /* set string1 to 0 */
```

SEE ALSO
memcpy

FUNCTION
modf—separate integral and fractional parts

SYNOPSIS
```
#include <math.h>
double modf(double, double *);
```

DESCRIPTION
The `modf` function separates the first argument into integral and fractional portions. The fractional portion is returned and the integral portion is stored in the object pointed to by `*`. The integral and fractional portions have the same sign as the input.

ERROR CONDITIONS
The `modf` function does not return an error condition.

EXAMPLE
```
#include <math.h>
double y, n;

y = modf(-12.345, &n); /* y = -12.0, n = -0.345 */
```

SEE ALSO
`frexp`

FUNCTION

pow—raise to a power

SYNOPSIS

```
#include <math.h>
double pow(double, double);
```

DESCRIPTION

The `pow` function computes the value of the first argument raised to the power of the second argument.

The `pow` function returns 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

A domain error occurs if the first argument is negative and the second argument cannot be represented as an integer. If the first argument is zero, the second argument is less than or equal to zero and the result cannot be represented, `EDOM` is stored in `errno`.

EXAMPLE

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

z = pow(4.0, 2.0);          /* z = 16.0 */
```

SEE ALSO

`ldexp`

FUNCTION
raise—force a signal

SYNOPSIS
```
#include <signal.h>
int raise(int sig);
```

DESCRIPTION
The `raise` function sends the signal `sig` to the executing program. The `raise` function forces interrupts wherever possible and simulates an interrupt otherwise. The `sig` argument must be one of the following signals listed below in priority order:

Signal Value	*Meaning*	
SIG_IGN	Signal Ignore	
SIG_DFL	Signal Default	
SIG_ERR	Signal Error	
SIGINT3	Interrupt 3	<ADSP-2100 Only>
SIGINT2	Interrupt 2	
SIGINT1	Interrupt 1	
SIGINT0	Interrupt 0	
SIGSPORT0XMIT	Signal Sport 0 Transmit	<Not ADSP-2100>
SIGSPORT0RECV	Signal Sport 0 Receive	
SIGTIMER	Signal Timer	
SIGSPORT1XMIT	Signal Sport 1 Transmit	<Not ADSP-2100>
SIGSPORT1RECV	Signal Sport 1 Receive	<Not ADSP-2100>
SIGHIPWRITE	Signal HIP Write	<ADSP-2111, ADSP-2150, ADSP-2171 Only>
SIGHIPREAD	Signal HIP Read	<ADSP-2111, ADSP-2150, ADSP-2171 Only>
SIGPOWERDOWN	Signal Power Down	<ADSP-2150, ADSP-2171>
SIGCODECXMIT	Signal Codec Transmit	<ADSP-2150 Only>
SIGCODECREC	Signal Codec Record	<ADSP-2150 Only>
SIGSWI1	Signal Software Interupt 1	<ADSP-2171 Only >
SIGSWI0	Signal Software Interupt 0	<ADSP-2171 Only >

Note: Interrupts are nested by default.

ERROR CONDITIONS

The `raise` function returns a zero if successful, a nonzero value if it fails.

EXAMPLE

```
#include <signal.h>

raise(SIG_DFL);/* invoke the signal default */
```

SEE ALSO

`interrupt, signal`

FUNCTION

signal—define signal handling

SYNOPSIS

```
#include <signal.h>
void (*signal(int sig, void (*func)(int))) (int);
```

DESCRIPTION

The `signal` function determines how a signal received during program execution is handled. It causes a single occurance of an interrupt to be responded to. The `sig` argument must be one of the following values that are listed below in highest to lowest priority of interrupts:

Signal Value	*Meaning*	
SIG_IGN	Signal Ignore	
SIG_DFL	Signal Default	
SIG_ERR	Signal Error	
SIGINT3	Interrupt 3	<ADSP-2100 Only>
SIGINT2	Interrupt 2	
SIGINT1	Interrupt 1	
SIGINT0	Interrupt 0	
SIGSPORT0XMIT	Signal Sport 0 Transmit	<Not ADSP-2100>
SIGSPORT0RECV	Signal Sport 0 Receive	
SIGTIMER	Signal Timer	
SIGSPORT1XMIT	Signal Sport 1 Transmit	<Not ADSP-2100>
SIGSPORT1RECV	Signal Sport 1 Receive	<Not ADSP-2100>
SIGHIPWRITE	Signal HIP Write	<ADSP-2111, ADSP-2150, ADSP-2171 Only>
SIGHIPREAD	Signal HIP Read	<ADSP-2111, ADSP-2150, ADSP-2171 Only>
SIGPOWERDOWN	Signal Power Down	<ADSP-2150, ADSP-2171>
SIGCODECXMIT	Signal Codec Transmit	<ADSP-2150 Only>
SIGCODECREC	Signal Codec Record	<ADSP-2150 Only>
SIGSWI1	Signal Software Interupt 1	<ADSP-2171 Only >
SIGSWI0	Signal Software Interupt 0	<ADSP-2171 Only >

The `signal` function causes the receipt of the signal number `func` to be handled in one of the following ways:

`func` value	*action*
`SIG_DFL`	The default action is taken.
`SIG_IGN`	The signal is ignored.
`Function address`	The function pointed to by `func` is executed.

The function pointed to by `func` is executed once when the signal is received. Handling is then returned to the default state.

Note: Interrupts are nested by default.

ERROR CONDITIONS

The `signal` function returns `SIG_ERR` and sets `errno` to `SIG_ERR` if it does not recognize the requested signal.

EXAMPLE

```
signal(SIG_TIMER, int_handler);
/*Executes the function int_handler() upon */
/*receipt of a timer interrupt */
```

SEE ALSO

`interrupt, raise`

FUNCTION
sin—sine

SYNOPSIS
```
#include <math.h>
double sin(double);
```

DESCRIPTION
The `sin` function returns the sine of `x`. The input is interpreted as a radian; the output is in the range [-1, 1].

The `sin` function returns 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.

Although the `sin` function accepts input over the entire floating-point range, the accuracy of the result decreases significantly for an input greater than $\pi^{12}/2$.

ERROR CONDITIONS
The `sin` function does not return an error condition.

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

y = sin(3.14159);      /* y = 0.0 */
```

SEE ALSO
asin, cos

FUNCTION
sinh—hyperbolic sine

SYNOPSIS
```
#include <math.h>
double sinh(double);
```

DESCRIPTION
The `sinh` functions return the hyperbolic sine of `x` and 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
For input values greater than 2^{12}, the `sinh` function returns `HUGE_VAL` and sets `errno` to `ERANGE` to indicate overflow.

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

y = sinh(x);
```

SEE ALSO
`cosh`

FUNCTION
sqrt—square root

SYNOPSIS
```
#include <math.h>
double sqrt(double);
```

DESCRIPTION
The `sqrt` function returns the positive square root of `x` and 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 `sqrt` function returns a zero for a negative input and sets `errno` to `EDOM` to indicate a domain error.

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

y = sqrt(2.0);    /* y = 1.414..... */
```

SEE ALSO
No references to this function.

FUNCTION

strcat—concatenate strings

SYNOPSIS

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

DESCRIPTION

The `strcat` function appends a copy of the null-terminated string pointed to by `s2` to the end of the null-terminated string pointed to by `s1`. It returns a pointer to the new `s1` string, which is null-terminated.

The behavior of `strcat` is undefined if the two strings overlap.

ERROR CONDITIONS

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

EXAMPLE

```
#include <string.h>
char string1[50];

string1[0] = 'A';
string1[1] = 'B';
string1[2] = '\0';
strcat(string1, "CD"); /* new string is "ABCD" */
```

SEE ALSO

strncat

FUNCTION
strcmp—compare strings

SYNOPSIS
```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

DESCRIPTION
The `strcmp` function compares (lexicographically) the null-terminated strings pointed to by `s1` and `s2`. It returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the `s2` string is greater than the `s1` string, and a zero if the strings are the same.

ERROR CONDITIONS
The `strcmp` function does not return an error condition.

EXAMPLE
```
#include <string.h>
char string1[50], string2[50];

if (strcmp(string1, string2))
printf("%s is different than %s \n", string1,
string2);
```

SEE ALSO
memcmp, strncmp

FUNCTION

strcpy—copy from one string to another

SYNOPSIS

```
#include <string.h>
void *strcpy(char *, _const_ char *);
```

DESCRIPTION

The `strcpy` function copies the null-terminated string pointed to by `s2` into the space pointed to by `s1`. Memory allocated for `s1` must be large enough to hold `s2`, plus one space for the null character (‘\0’). The behavior of `strcpy` is undefined if the two objects overlap or if `s1` is not large enough. The `strcpy` function returns the new `s1`.

ERROR CONDITIONS

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

EXAMPLE

```
#include <string.h>
char string1[50];

strcpy(string1, “SOMEFUN”);
/* SOMEFUN is copied into string1 */
```

SEE ALSO

memcpy, strncpy

FUNCTION
strlen—string length

SYNOPSIS
```
#include <string.h>
size_t strlen(const char *s1);
```

DESCRIPTION
The `strlen` function returns the length of the null-terminated string pointed to by `s` (not including the null).

ERROR CONDITIONS
The `strlen` function does not return an error condition.

EXAMPLE
```
#include <string.h>
size_t len;

len = strlen(“SOMEFUN”);    /* len = 7 */
```

SEE ALSO
No references to this function.

FUNCTION

strncat—concatenate characters from one string to another

SYNOPSIS

```
#include <string.h>
char *strncat(char *s1, const char *s2,
     size_t n);
```

DESCRIPTION

The `strncat` function appends a copy of up to `n` characters in the null-terminated string pointed to by `s2` to the end of the null-terminated string pointed to by `s1`. It returns a pointer to the new `s1` string.

The behavior of `strncat` is undefined if the two strings overlap. The new `s1` string is terminated with a null ('\0').

ERROR CONDITIONS

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

EXAMPLE

```
#include <string.h>
char string1[50], *ptr;

string1[0]='\0';
strncat(string1, "MOREFUN", 4);
/* string1 equals "MORE" */
```

SEE ALSO

strcat

FUNCTION
strncmp—compare characters in strings

SYNOPSIS
```
#include <string.h>
int strncmp(const char *s1, const char *s2,
          size_t n);
```

DESCRIPTION
The `strncmp` function compares (lexicographically) up to `n` characters of the null-terminated strings pointed to by `s1` and `s2`. It returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the `s2` string is greater than the `s1` string, and a zero if the strings are the same.

ERROR CONDITIONS
The `strncmp` function does not return an error condition.

EXAMPLE
```
#include <string.h>
char *ptr1;

ptr1 = “TEST1”;
if (strncmp(ptr1, “TEST”, 4) ==0)
printf(“%s starts with TEST \n”, ptr1);
```

SEE ALSO
memcmp, strcmp

FUNCTION

strncpy—copy characters from one string to another

SYNOPSIS

```
#include <string.h>
char *strncpy(char *s1, const char *s2,
     size_t n);
```

DESCRIPTION

The `strncpy` function copies up to `n` characters of the null-terminated string pointed to by `s2` into the space pointed to by `s1`. If the last character copied from `s2` is not a null, the result will not end with a null.

The behavior of `strncpy` is undefined if the two objects overlap. The `strncpy` function returns the new `s1`.

If the `s2` string contains fewer than `n` characters, the `s1` string is padded with nulls until all `n` characters have been written.

ERROR CONDITIONS

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

EXAMPLE

```
#include <string.h>
char string1[50];

strncpy(string1, “MOREFUN”, 4);
/* MORE is copied into string1 */
string1[4] = ‘/0’;  /* must null-terminate string1
       */
```

SEE ALSO

memcpy, strcpy

FUNCTION
tan—tangent

SYNOPSIS
```
#include <math.h>
double tan(double);
```

DESCRIPTION
The `tan` function returns 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 `tan` function does not return an error condition.

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

y = tan(3.14159/4.0);  /* y = 1.0 */
```

SEE ALSO
atan, atan2

FUNCTION

tanh—hyperbolic tangent

SYNOPSIS

```
#include <math.h>
double tanh(double);
```

DESCRIPTION

The `tanh` function returns 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 `tanh` function does not return an error condition.

EXAMPLE

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

y = tanh(x);
```

SEE ALSO

sinh, cosh

FUNCTION
timer_off—disable ADSP-21xx timer

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

DESCRIPTION
This function is an Analog Devices extension to the ANSI standard.

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

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

EXAMPLE
```
#include <misc.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`

FUNCTION
timer_on—enable ADSP-21xx timer

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

DESCRIPTION
This function is an Analog Devices extension to the ANSI standard.

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

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

EXAMPLE
```
#include <misc.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`

FUNCTION
timer_set—initialize ADSP-21xx timer

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

DESCRIPTION
This function is an Analog Devices extension to the ANSI standard.

The `timer_set` function sets the ADSP-21xx timer registers `TPERIOD` and `TCOUNT`. The function returns a 1 if the timer is enabled, a 0 if the timer is disabled.

The `TSCALE` value is used to set the `TSCALE` register. For a complete description of the ADSP-21xx timer, refer to the *ADSP-2100 Family User's Guide.*

Note: 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 initializing overhead that could create an infinite loop.

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

EXAMPLE
```
#include <misc.h>

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

SEE ALSO
`timer_on, timer_off`

FUNCTION

va_arg—get next argument in variable list

SYNOPSIS

```
#include <stdarg.h>
void va_arg(va_list ap, type);
```

DESCRIPTION

The `va_arg` macro can only be used after the `va_start` macro has been invoked. The `va_arg` macro uses the pointer initialized by `va_start` to return the value and type of the next argument in the list of optional arguments. It then increments the pointer. The header file `stdarg.h` defines a pointer type called `va_list` that is used to access the list of variable arguments.

The parameter `type` is a type name such that if a `*` is appended to it, then the type of a pointer to an object of type `void` can be obtained. If there is no `next` variable, then there is no defined behavior for `va_arg`.

ERROR CONDITIONS

The `va_arg` macro does not return an error condition.

EXAMPLE

```
#include <stdarg.h>
double y;
va_list ap;

y=va_arg(ap,double);
     /* y is equal to the next argument in ap */
```

SEE ALSO

va_start, va_end

FUNCTION

va_end—reset variable list pointer

SYNOPSIS

```
#include <stdarg.h>
void va_end(va_list ap);
```

DESCRIPTION

The `va_end` macro can only be used after the `va_start` macro has been invoked. The `va_end` macro resets the pointer initialized by `va_start`. This is necessary for a proper return from the function in which the macros are invoked.

ERROR CONDITIONS

The `va_end` macro does not return an error condition.

EXAMPLE

```
#include <stdarg.h>
va_list ap;

va_end(ap);
```

SEE ALSO

va_arg, va_start

FUNCTION

va_start—set variable list pointer

SYNOPSIS

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

DESCRIPTION

The `va_start` macro should be invoked in a function that requires a variable number of arguments. The `va_start` macro initializes a pointer of type `va_list` to point to an argument that immediately follows the last required (named) parameter in the functions argument list. This last required parameter is provided as an argument in the invocation of `va_start`. The `va_start` macro must be invoked before either the `va_arg` or `va_end` macro can be invoked.

ERROR CONDITIONS

The `va_start` macro does not return an error condition.

EXAMPLE

```
#include <stdarg.h>
va_list ap;

va_start(ap, last_required_argument);
```

SEE ALSO

va_arg, va_end