VISUALDSP++® 5.0
Run-Time Library Manual for SHARC® Processors

Revision 1.5, January 2011

Part Number
82-000420-09

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

ANALOG
DEVICES

Copyright Information

Printed in the USA.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

The Analog Devices logo, Blackfin, EZ-KIT Lite, SHARC, TigerSHARC, and VisualDSP++ are registered trademarks of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.

CONTENTS

DSP RUN-TIME LIBRARY

PREFACE

Thank you for purchasing Analog Devices, Inc. development software for signal processing applications.

Purpose of This Manual

The *VisualDSP++ 5.0 Run-Time Library Manual for SHARC Processors* contains information about the C/C++ and DSP run-time libraries for SHARC® (ADSP-21xxx) processors. It leads you through the process of using library routines and provides information about the ANSI standard header files and different libraries that are included with this release of the `cc21k` compiler.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. This manual assumes that the audience has a working knowledge of the SHARC architecture and the C/C++ programming languages.

Programmers who are unfamiliar with SHARC processors can use this manual, but they should supplement it with other texts (such as the appropriate hardware reference and programming reference manuals) that describe their target architecture.

Manual Contents

This manual contains:

- Chapter 1, "C/C++ Run-Time Library"
  Describes how to use library functions and provides a complete C/C++ library function reference (for functions covered in the current compiler release)

- Chapter 2, "DSP Run-Time Library"
  Describes how to use DSP library functions and provides a complete library function reference (for functions covered in the current compiler release)

What's New in This Manual

This revision (1.5) of the *VisualDSP++ 5.0 Run-Time Library Manual for SHARC Processors* documents changes/additions related to the run-time library for VisualDSP++® 5.0 and subsequent updates (up to update 9). Changes to this book from revision 1.4 include:

- The library now supports the 64-bit integer types `long long` and `unsigned long long`. The following new functions have been added: `atoll`, `llabs`, `llavg`, `llclip`, `llcount_ones`, `lldiv`, `llmax`, `llmin`, `strtoll`, `strtoull`.

- The library now supports the native fixed-point `fract` type. The following new functions have been added: `absfx`, `bitsfx`, `countlsfx`, `divifx`, `fxbits`, `fxdivi`, `idivfx`, `mulifx`, `roundfx`, `strtofxfx`.

- Corrections of typographic errors and reported document errata

This manual documents C/C++ and DSP libraries for all current SHARC processors listed in the online help.

Refer to the *VisualDSP++ 5.0 C/C++ Compiler Manual* for a complete description of C/C++ compiler features and the use of the `cc21k` compiler in developing efficient and user-friendly source code.

Technical or Customer Support

You can reach Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at `http://www.analog.com/processors/technical_support`
- E-mail tools questions to `processor.tools.support@analog.com`
- E-mail processor questions to
  `processor.support@analog.com (World wide support)`
  `processor.europe@analog.com (Europe support)`
  `processor.china@analog.com (China support)`
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor

Supported Processors

The name *SHARC* refers to a family of Analog Devices, Inc. high-performance 32-bit floating-point digital signal processors that can be used in speech, sound, graphics, and imaging applications. For a complete list of processors supported by VisualDSP++ 5.0, refer to VisualDSP++ online Help.

Product Information

Product information can be obtained from the Analog Devices Web site, VisualDSP++ online Help system, and a technical library CD.

Analog Devices Web Site

The Analog Devices Web site, `www.analog.com`, provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

To access a complete technical library for each processor family, go to `http://www.analog.com/processors/technical_library`. The manuals selection opens a list of current manuals related to the product as well as a link to the previous revisions of the manuals. When locating your manual title, note a possible errata check mark next to the title that leads to the current correction report against the manual.

Also note, `MyAnalog.com` is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information about products you are interested in. You can choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests, including documentation errata against all manuals. `MyAnalog.com` provides access to books, application notes, data sheets, code examples, and more.

Visit `MyAnalog.com` to sign up. If you are a registered user, just log on. Your user name is your e-mail address.

VisualDSP++ Online Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, Dinkum Abridged C++ library, and FLEXnet License Tools documentation. You can search easily across the entire VisualDSP++ documentation set for any topic of interest.

For easy printing, supplementary Portable Documentation Format (`.pdf`) files for all manuals are provided on the VisualDSP++ installation CD.

Each documentation file type is described as follows.

File	Description
`.chm`	Help system files and manuals in Microsoft help format
`.htm` or `.html`	Dinkum Abridged C++ library and FLEXnet license tools software documentation. Viewing and printing the `.html` files requires a browser, such as Internet Explorer 6.0 (or higher).
`.pdf`	VisualDSP++ and processor manuals in PDF format. Viewing and printing the `.pdf` files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

Technical Library CD

The technical library CD contains seminar materials, product highlights, a selection guide, and documentation files of processor manuals, VisualDSP++ software manuals, and hardware tools manuals for the following processor families: Blackfin®, SHARC, TigerSHARC®, ADSP-218x, and ADSP-219x.

To order the technical library CD, go to `http://www.analog.com/processors/technical_library`, navigate to the manuals page for your processor, click the request CD check mark, and fill out the order form.

Data sheets, which can be downloaded from the Analog Devices Web site, change rapidly, and therefore are not included on the technical library CD. Technical manuals change periodically. Check the Web site for the latest manual revisions and associated documentation errata.

EngineerZone

EngineerZone is a technical support forum from Analog Devices. It allows you direct access to ADI technical support engineers. You can search FAQs and technical information to get quick answers to your embedded processing and DSP design questions.

Use EngineerZone to connect with other DSP developers who face similar design challenges. You can also use this open forum to share knowledge and collaborate with the ADI support team and your peers. Visit `http://ez.analog.com` to sign up.

Social Networking Web Sites

You can now follow Analog Devices SHARC development on Twitter and LinkedIn. To access:

- Twitter: `http://twitter.com/ADISHARC`
- LinkedIn: Network with the LinkedIn group, Analog Devices SHARC: `http://www.linkedin.com`

Notation Conventions

Text conventions used in this manual are identified and described as follows.

Additional conventions, which apply only to specific chapters, may appear throughout this document.

Example	Description
Close command (**File** menu)	Titles in in bold style reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
	Note: For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	Caution: Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
	Warning: Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

1 C/C++ RUN-TIME LIBRARY

The C and C++ run-time libraries are collections of functions, macros, and class templates that you can call from your source programs. Many functions are implemented in the ADSP-21xxx assembly language. C and C++ programs depend on library functions to perform operations that are basic to the C and C++ programming environments. These operations include memory allocations, character and string conversions, and math calculations. Using the library simplifies your software development by providing code for a variety of common needs.

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

- "C and C++ Run-Time Libraries Guide" on page 1-2
  provides introductory information about the ANSI/ISO standard C and C++ libraries. It also provides information about the ANSI standard header files and built-in functions that are included with this release of the `cc21k` compiler.
- "C Run-Time Library Reference" on page 1-79
  contains reference information about the C run-time library functions included with this release of the `cc21k` compiler.

The `cc21k` compiler provides a broad collection of library functions, including those required by the ANSI standard and additional functions supplied by Analog Devices that are of value in signal processing applications. In addition to the standard C library, this release of the compiler software includes the Abridged C++ library, a conforming subset of the standard C++ library. The Abridged C++ library includes the embedded C++ and embedded standard template libraries.

This chapter describes the standard C/C++ library functions that are supported in the current release of the run-time libraries. Chapter 2, "DSP Run-Time Library" describes a number of signal processing, matrix, and statistical functions that assist code development.

For more information on the algorithms on which many of the C library's math functions are based, see W. J, Cody and W. Waite, *Software Manual for the Elementary Functions*, Englewood Cliffs, New Jersey: Prentice Hall, 1980. For more information on the C++ library portion of the ANSI/ISO Standard for C++, see Plauger, P. J. (Preface), *The Draft Standard C++ Library,* Englewood Cliffs, New Jersey: Prentice Hall, 1994, (ISBN: 0131170031).

The Abridged C++ library software documentation is located on the VisualDSP++ installation CD in the <install_path>\Docs\Reference folder. Viewing or printing these files requires a browser, such as Internet Explorer 6.0 (or higher). You can copy these files from the installation CD onto another disk.

C and C++ Run-Time Libraries Guide

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

- "Calling Library Functions" on page 1-3
- "Linking Library Functions" on page 1-4
- "Library Attributes" on page 1-13
- "Working With Library Header Files" on page 1-18
- "Calling Library Functions From an ISR" on page 1-37

- "Using the Libraries in a Multi-Threaded Environment" on page 1-38
- "Using Compiler Built-In C Library Functions" on page 1-39
- "Abridged C++ Library Support" on page 1-41
- "Measuring Cycle Counts" on page 1-48
- "File I/O Support" on page 1-59

For information on the C library's contents, see "C Run-Time Library Reference" on page 1-79. For information on the Abridged C++ library's contents, see "Abridged C++ Library Support" on page 1-41.

Calling Library Functions

To use a C/C++ library function, call the function by name and give the appropriate arguments. The name and arguments for each function appear on the function's reference page. The reference pages appear in the "C Run-Time Library Reference" on page 1-79.

Like other functions you use, library functions should be declared. Declarations are supplied in header files. For more information about the header files, see "Working With Library Header Files" on page 1-18.

Function names are C/C++ function names. If you call a C/C++ run-time library function from an assembly program, you must use the assembly version of the function name (prefix an underscore on the name). For more information on the naming conventions, see Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, in the section "C/C++and Assembly Interface."

> You can use the archiver, `elfar`, described in the *VisualDSP++ 5.0 Linker and Utilities Manual*, to build library archive files of your own functions.

Linking Library Functions

The C/C++ run-time library is organized as five libraries:

- C run-time library – Comprises all the functions that are defined by the ANSI standard
- C++ run-time library
- DSP run-time library – Contains additional library functions supplied by Analog Devices that provide services commonly required by DSP applications
- I/O library – Supports a subset of the C standard's I/O functionality
- Fixed-point I/O library – Supports a subset of the C standard's I/O functionality for fractional data types

In general, several versions of the C/C++ run-time library are supplied in binary form; for example, variants are available for different SHARC architectures and are listed in Table 1-1, Table 1-2, Table 1-3, and Table 1-4. Some versions of these binary files are also built for running in a multi-threaded environment; these binaries have `mt` in their filename.

In addition to regular run-time libraries, VisualDSP++ 5.0 also has `libio*_lite.dlb` libraries which provide smaller versions of `LibIO` (the I/O run-time support library) with more limited functionality. These smaller `LibIO` libraries can be used by specifying the switch `-flags-link -MD__LIBIO_LITE` on the build command line. There are also `libio*_fx.dlb` libraries which provide versions of `LibIO` (the I/O run-time support library) with full support for the fixed-point format specifiers for the `fract` types. These libraries can be used by specifying the switch `-flags-link -MD__LIBIO_FX` on the build command line.

Table 1-1 contains a list of the run-time libraries and start-up files that have been built for the ADSP-21020 and ADSP-2106x processors, and are installed in the subdirectory `21k\lib`.

Table 1-1. C/C++ Files and Libraries for ADSP-210xx Processors

Description	Library Name	Comments

C run-time library	`libc.dlb`
	`libc020.dlb`
	`libcmt.dlb`
		
		ADSP-21020 processor only

C++ run-time library	`libcpp.dlb` `libcppmt.dlb`	
C++ run-time library with exception handling support	`libcppeh.dlb` `libcppehmt.dlb`	

Legacy library	`libcpprt.dlb`
	`libcpprtmt.dlb`
	`libcpprteh.dlb`
	`libcpprtehmt.dlb`
	`libeh.dlb`
	`libehmt.dlb`
		These libraries contain no functions and are only provided for the purpose of linking against a legacy `.ldf` file

DSP run-time library	`libdsp.dlb` `libdsp020.dlb`	 ADSP-21020 processor only

I/O run-time library	`libio.dlb`
	`libio020.dlb`
	`libiomt.dlb`
		
		ADSP-21020 processor only

I/O run-time library with no support for alternative device drivers or printf("%a")	`libio_lite.dlb`
	`libio020_lite.dlb`
	`libio_litemt.dlb`
	
	`libio32.dlb`
	`libio64.dlb`
		
		ADSP-21020 processor only
		
		Legacy library
		Legacy library

I/O run-time library with full support for the fixed-point format specifiers	`libio_fx.dlb` `libio_fxmt.dlb`	

Table 1-1. C/C++ Files and Libraries for ADSP-210xx Processors (Cont'd)

Description	Library Name	Comments

C start-up file — calls set-up routines and `main()`	`020_hdr.doj`
	`060_hdr.doj`
	
	`061_hdr.doj`
	`065L_hdr.doj`
		ADSP-21020 processor only
		ADSP-21060/062 processors only
		ADSP-21061 processor only
		ADSP-21065L processor only

C start-up file with EZ-KIT — calls set-up routines and `main()`	`061_hdr_ezkit.doj` `065L_hdr_ezkit.doj`	ADSP-21061 processor only ADSP-21065L processor only

C++ start-up file — calls set-up routines and `main()`	`060_cpp_hdr.doj`
	
	`061_cpp_hdr.doj`
	`065L_cpp_hdr.doj`
	
	`060_cpp_hdr_mt.doj`
	
	`061_cpp_hdr_mt.doj`
	`065L_cpp_hdr_mt.doj`
		ADSP-21060/062 processors only
		ADSP-21061 processor only
		ADSP-21065L processor only
		
		ADSP-21060/062 processors only
		ADSP-21061 processor only
		ADSP-21065L processor only

C++ start-up file with EZ-KIT — calls set-up routines and `main()`	`061_cpp_hdr_ezkit.doj`
	`065L_cpp_hdr_ezkit.doj`
	
	`061_cpp_hdr_ezkit_mt.doj`
	`065L_cpp_hdr_ezkit_mt.doj`
		ADSP-21061 processor only
		ADSP-21065L processor only
		
		ADSP-21061 processor only
		ADSP-21065L processor only

The binary files that have been built for ADSP-2116x processors are catalogued in Table 1-2.

Table 1-2. C/C++ Files and Libraries for ADSP-2116x Processors

Description	Library Name	Comments

C run-time library	`libc160.dlb`
	`libc161.dlb`
	
	`libc160mt.dlb`
	`libc161mt.dlb`
		ADSP-21160 processor only
		ADSP-21161 processor only
		
		ADSP-21160 processor only
		ADSP-21161 processor only

C++ run-time library	`libcpp.dlb` `libcppmt.dlb`	

Table 1-2. C/C++ Files and Libraries for ADSP-2116x Processors (Cont'd)

Description	Library Name	Comments
C++ run-time library with exception handling support	`libcppeh.dlb` `libcppehmt.dlb`	

Legacy library	`libcpprt.dlb`
	`libcpprtmt.dlb`
	`libcpprteh.dlb`
	`libcpprtehmt.dlb`
	`libeh.dlb`
	`libehmt.dlb`
		These libraries contain no functions and are only provided for the purpose of linking against a legacy `.ldf` file

DSP run-time library	`libdsp160.dlb`	
I/O run-time library	`libio.dlb` `libiomt.dlb`	

I/O run-time library with no support for alternative device drivers or printf("%a")	`libio_lite.dlb`
	`libio_litemt.dlb`
	
	`libio32.dlb`
	`libio64.dlb`
		
		Legacy library
		Legacy library

I/O run-time library with full support for the fixed-point format specifiers	`libio_fx.dlb` `libio_fxmt.dlb`	
C start-up file — calls set-up routines and `main()`	`160_hdr.doj` `161_hdr.doj`	ADSP-21160 processor only ADSP-21161 processor only
C start-up file with EZ-KIT — calls set-up routines and `main()`	`160_hdr_ezkit.doj`	ADSP-21160 processor only

C++ start-up file — calls set-up routines and `main()`	`160_cpp_hdr.doj`
	`161_cpp_hdr.doj`
	
	`160_cpp_hdr_mt.doj`
	`161_cpp_hdr_mt.doj`
		ADSP-21160 processor only
		ADSP-21161 processor only
		
		ADSP-21160 processor only
		ADSP-21161 processor only

C++ start-up file with EZ-KIT — calls set-up routines and `main()`	`160_cpp_hdr_ezkit.doj` `160_cpp_hdr_ezkit_mt.doj`	ADSP-21160 processor only ADSP-21160 processor only

The run-time libraries and binary files for the ADSP-21160 processors in this table have been compiled with the `-workaround rframe` compiler switch, while those for the ADSP-21161 processors have been compiled with the `-workaround 21161-anomaly-45` switch. An additional set of libraries and binary files that also work around the shadow write FIFO anomaly that affect ADSP-2116x chips is installed in the subdirectory `211xx\lib\swfa`.

Table 1-3 contains a list and a brief description of the library files that have been built for the ADSP-212xx processors. These files are installed in the subdirectory `212xx\lib`.

Table 1-3. C/C++ Libraries for ADSP-212xx Processors

Description	Library Name	Comments
C run-time library	`libc26x.dlb` `libc26xmt.dlb`	
C++ run-time library	`libcpp.dlb` `libcppmt.dlb`	
C++ run-time library with exception handling support	`libcppeh.dlb` `libcppehmt.dlb`	

Legacy library	`libcpprt.dlb`
	`libcpprtmt.dlb`
	`libcpprteh.dlb`
	`libcpprtehmt.dlb`
	`libeh.dlb`
	`libehmt.dlb`
		These libraries contain no functions and are only provided for the purpose of linking against a legacy `.ldf` file

DSP run-time library	`libdsp26x.dlb`	
I/O run-time library	`libio.dlb` `libiomt.dlb`	
I/O run-time library with no support for alternative device drivers or `printf("%a")`	`libio_lite.dlb` `libio_litemt.dlb`	
I/O run-time library with full support for the fixed-point format specifiers	`libio_fx.dlb` `libio_fxmt.dlb`	

The libraries located in `212xx\lib` are built without any workarounds enabled. There are directories within the `212xx\lib` directory named `2126x_rev_<revision>` that contain libraries built for that specific revision, for example, `2126x_rev_0.0`. A single revision library directory may support more than one specific silicon revision; as an example, `2126x_rev_0.0` supports revisions 0.0, 0.1 and 0.2 of ADSP-2126x processors.

In addition, a library directory called `2126x_any` is supplied. Libraries in this directory will contain workarounds for all relevant anomalies on all revisions of ADSP-2126x processors.

The `-si-revision` switch can be used to specify a silicon revision—VisualDSP++ will use the appropriate libraries to build the application.

As well as libraries, the directory `212xx\lib` also contains two different sets of object files. The first set of object files are the C start-up files for the ADSP-212*xx* processor family. Each processor in the family has its own C start-up file that initializes the environment and then calls `main()`. These object files have names of the form `2`*`xx`*`_hdr.doj` where *xx* identifies a specific processor; for example, the file `261_hdr.doj` is the C start-up file for the ADSP-21261 processor.

The second set of object files in the directory `212xx\lib` are the start-up files for C++ applications; they have names of the form `2`*`xx`*`_cpp_hdr.doj` and `2`*`xx`*`_cpp_hdr_mt.doj`, where *xx* represents a specific ADSP-212*xx* processor. For example, the file `261_cpp_hdr.doj` initializes the run-time environment and then calls `main()`, for a C++ application that runs on the ADSP-21261 processor.

Table 1-4 describes the library files that have been built for the ADSP-213xx processors, and which are installed in the subdirectory `213xx\lib`.

Table 1-4. C/C++ Libraries for ADSP-213xx Processors

Description	Library Name	Comments

C run-time library	`libc36x.dlb`
	`libc36xmt.dlb`
	`libc37x.dlb`
	`libc37xmt.dlb`	

C++ run-time library	`libcpp.dlb` `libcppmt.dlb`	
C++ run-time library with exception handling support	`libcppeh.dlb` `libcppehmt.dlb`	

Legacy library	`libcpprt.dlb`
	`libcpprtmt.dlb`
	`libcpprteh.dlb`
	`libcpprtehmt.dlb`
	`libeh.dlb`
	`libehmt.dlb`
		These libraries contain no functions and are only provided for the purpose of linking against a legacy `.ldf` file.

DSP run-time library	`libdsp36x.dlb` `libdsp37x.dlb`	
I/O run-time library	`libio.dlb` `libiomt.dlb`	
I/O run-time library with no support for alternative device drivers or `printf("%a")`	`libio_lite.dlb` `libio_litemt.dlb`	
I/O run-time library with full support for the fixed-point format specifiers	`libio_fx.dlb` `libio_fxmt.dlb`	

As well as libraries, the directory `213xx\lib` also contains two different sets of object files. The first set of object files are the C start-up files for the ADSP-213*xx* processor family. Each processor in the family has its own C start-up file that initializes the environment and then calls `main()`. These object files have names of the form `3xx_hdr.doj` where *xx* identifies a specific processor; for example, the file `363_hdr.doj` is the C start-up file for the ADSP-21363 processor.

The second set of object files in the directory `213xx\lib` are the start-up files for C++ applications; they have names of the form `3xx_cpp_hdr.doj` and `3xx_cpp_hdr_mt.doj`, where *xx* represents a specific ADSP-212*xx* processor. For example, the file `363_cpp_hdr.doj` initializes the run-time environment and then calls `main()`, for a C++ application that runs on the ADSP-21363 processor.

Table 1-5 contains a list and a brief description of the library files that have been built for the ADSP-214xx processors. These files are installed in the subdirectory `214xx\lib`. The libraries are built in short-word mode by default, though there are versions which have been built in normal-word mode; these binaries have `nwc` in their filename.

Table 1-5. C/C++ Libraries for ADSP-214xx Processors

Description	Library Name	Comments

C run-time library	`libc.dlb`
	`libcmt.dlb`
	`libc_nwc.dlb`
	`libcmt_nwc.dlb`	

C++ run-time library	`libcpp.dlb`
	`libcppmt.dlb`
	`libcpp_nwc.dlb`
	`libcppmt_nwc.dlb`	

C++ run-time library with exception handling support	`libcppeh.dlb`
	`libcppehmt.dlb`
	`libcppeh_nwc.dlb`
	`libcppehmt_nwc.dlb`	

DSP run-time library	`libdsp.dlb` `libdsp_nwc.dlb`	

Table 1-5. C/C++ Libraries for ADSP-214xx Processors (Cont'd)

Description	Library Name	Comments
I/O run-time library	libio.dlb libiomt.dlb libio_nwc.dlb libiomt_nwc.dlb	
I/O run-time library with no support for alternative device drivers or printf("%a")	libio_lite.dlb libio_litemt.dlb libio_lite_nwc.dlb libio_litemt_nwc.dlb	
I/O run-time library with full support for the fixed-point format specifiers	libio_fx.dlb libio_fxmt.dlb libio_fx_nwc.dlb libio_fxmt_nwc.dlb	

The libraries located in `214xx\lib` are built without any workarounds enabled. In addition, a library directory called `21469_rev_any` is supplied. Libraries in this directory contain workarounds for all relevant anomalies on all revisions of ADSP-214xx processors.

As well as libraries, the directory `214xx\lib` also contains two different sets of object files. The first set of object files are the C start-up files for the ADSP-214*xx* processor family. Each processor in the family has its own C start-up file that initializes the environment and then calls `main()`. These object files have names of the form `214xx_hdr.doj` where *xx* identifies a specific processor; for example, the file `21462_hdr.doj` is the C start-up file for the ADSP-21462 processor.

The second set of object files in the directory `214xx\lib` are the start-up files for C++ applications; they have names of the form `214xx_cpp_hdr.doj` and `214xx_cpp_hdr_mt.doj`, where *xx* represents a specific ADSP-214*xx* processor. For example, the file `21462_cpp_hdr.doj` initializes the run-time environment and then calls `main()`, for a C++ application that runs on the ADSP-21462 processor.

When you call a 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 library's location is to use the default Linker Description File (ADSP-<your_target>.ldf).

If you are not using the default `.ldf` file, then either add the appropriate library/libraries to the `.ldf` file used for your project, or use the compiler's `-l` switch to specify the library to be added to the link line. For example, the switches `-lc -ldsp` add `libc.dlb` and `libdsp.dlb` to the list of libraries to be searched by the linker. For more information on the `.ldf` file, see the *VisualDSP++ 5.0 Linker and Utilities Manual.*

Library Attributes

The run-time libraries make use of file attributes. (See Chapter 1 of the VisualDSP++ 5.0 Compiler Manual for more details on how to use file attributes.) Each library function has a defined set of file attributes that are listed in Table 1-6. For each object `obj` in the run-time libraries the following is true.

Table 1-6. Run-Time Library Object Attributes

Attribute Name	Meaning of Attribute and Value
`libGroup`	A potentially multi-valued attribute. Each value is the name of a header file that either defines `obj`, or that defines a function that calls `obj`.
`libName`	The name of the library that contains `obj`, without the processor identifier. For example, suppose that `obj` were part of `libdsp160.dlb`, then the value of the attribute would be `libdsp`.
`libFunc`	The name of all the functions in `obj`. `libFunc` will have multiple values -both the C, and assembly linkage names will be listed. `libFunc` will also contain all the published C and assembly linkage names of objects in `obj`'s library that call into `obj`.

Table 1-6. Run-Time Library Object Attributes (Cont'd)

Attribute Name	Meaning of Attribute and Value
`prefersMem`	One of three values: `internal`, `external` or `any`. If `obj` contains a function that is likely to be application performance critical, it will be marked as `internal`. Most DSP run-time library functions fit into the `internal` category. If a function is deemed unlikely to be essential for achieving the necessary performance it will be marked as external (all the I/O library functions fall into this category). The default `.ldf` files use this attribute to place code and data optimally.
`prefersMemNum`	Analogous to `prefersMem` but takes a numeric string value. The attribute can be used in `.ldf` files to provide a greater measure of control over the placement of binary object files than is available using the `prefersMem` attribute. The values `"30"`, `"50"`, and `"70"` correspond to the `prefersMem` values `internal`, `any`, and `external` respectively. The default `.ldf` files use the `prefersMem` attribute in preference to the `prefersMemNum` attribute to specify the optimum placement of files.
`FuncName`	Multi-valued attribute whose values are all the assembler linkage names of the defined names in `obj`.

If an object in the run-time library calls into another object in the same library, whether it is internal or publicly visible, the called object will inherit extra `libGroup` and `libFunc` values from the caller.

The following example demonstrates how attributes would look in a small example library `libfunc.dlb` that comprises three objects: `func1.doj`, `func2.doj` and `subfunc.doj`. These objects are built from the following source modules:

File: `func1.h`

```
void func1(void);
```

File: `func2.h`

```
void func2(void);func1.c
```

```
#include func1.h"
void func1(void) {
   /* Compiles to func1.doj */
   subfunc();
}
```

File: `func2.c`

```
#include "func2.h"
void func2(void) {
   /* Compiles to func2.doj */
   subfunc();
}
```

File: `subfunc.c`

```
void subfunc(void) {
   /* Compiles to subfunc.doj */
}
```

The objects in `libfunc.dlb` have the attributes as defined in Table 1-7.

Table 1-7. Attribute Values in libfunc.dlb

Attribute	Value
func1.doj	
libGroup	func1.h
libName	libfunc
libFunc	_func1
libFunc	func1
FuncName	_func1
prefersMem	any([1])
prefersMemNum	50
func2.doj	
libGroup	func2.h
libName	libfunc
libFunc	_func2
libFunc	func2
FuncName	_func2
prefersMem	internal([2])
prefersMemNum	30
subfunc.doj	
libGroup	func1.h
libGroup	func2.h([3])
libName	libfunc
libFunc	_func1
libFunc	func1
libFunc	_func2
libFunc	func2
libFunc	_subfunc
libFunc	subfunc
FuncName	_subfunc
prefersMem	internal([4])
prefersMemNum	30

1 `func1.doj` will not be performance critical, based on its normal usage.
2 `func2.doj` will be performance critical in many applications, based on its normal usage.
3 `libGroup` contains the union of the `libGroup` attributes of the two calling objects.
4 `prefersMem` contains the highest priority of all the calling objects.

Exceptions to the Attribute Conventions

The library attribute convention has the following exceptions:
The C++ support libraries (libcpp*.dlb) all contain functions that have C++ linkage. Functions written in C++ have their function names encoded (often referred to as name mangling) to allow for the overloading of parameter types. The function name encoding includes all the parameter types, the return type and the namespace within which the function is declared. Whenever a function's name is encoded, the encoded name is used as the value for the libFunc attribute.

Table 1-8 lists additional libGroup attribute values.

Table 1-8. Additional libGroup Attribute Values

Value	Meaning
exceptions_support	Compiler support routines for C++ exceptions
floating_point_support	Compiler support routines for floating point arithmetic
integer_support	Compiler support routines for integer arithmetic
runtime_support	Other run-time functions that do not fit into any of the above categories
startup	One-time initialization functions called prior to the invocation of main

Objects with any of the libGroup attribute values listed in Table 1-8 will not contain any libGroup or libFunc attributes from any calling objects.

Table 1-9 presents a summary of the default memory placement using prefersMem.

Table 1-9. Default Memory Placement Summary

Library	Placement
libcpp*.dlb	any
idle*.doj libio*.dlb	external
libdsp*.dlb	internal except for the windowing functions and functions which generate a twiddle table which are external
libc*.dlb	any except for the stdio.h functions, which are external, and qsort, which is internal

Most of the functions contained within the DSP run-time library (libdsp*.dlb) have prefersMem=internal, because it is likely that any function called in this run-time library will make up a significant part of an application's cycle count.

Mapping Objects to FLASH Memory Using Attributes

When using the Memory Initializer to initialize code and data areas from flash memory, code and data used during the process of initialization must be mapped to flash memory to ensure it is available during boot-up. The requiredForROMBoot attribute is specified on library objects that contain such code and data and can be used in the .ldf file to perform the required mapping. See the *VisualDSP++ 5.0 Linker and Utilities Manual* for further information on memory initialization.

Working With Library Header Files

When you use a library function in your program, you should also include the function's header file with the #include preprocessor command. The header file for each function is identified in the **Synopsis** section of the function's reference page. Header files contain function prototypes. The compiler uses these prototypes to check that each function is called with the correct arguments.

A list of the header files that are supplied with this release of the `cc21k` compiler appears in Table 1-10. You should use a C standard text to augment the information supplied in this chapter.

Table 1-10. Standard C Run-Time Library Header Files

Header	Purpose	Standard
`adi_types.h`	Type definitions	Analog extension
`assert.h`	Diagnostics	ANSI
`ctype.h`	Character Handling	ANSI
`cycle_count.h`	Basic Cycle Counting	Analog extension
`cycles.h`	Cycle Counting with Statistics	Analog extension
`device.h`	Macros and data structures for alternative device drivers	Analog extension
`device_int.h`	Enumerations and prototypes for alternative device drivers	Analog extension
`errno.h`	Error Handling	ANSI
`float.h`	Floating Point	ANSI
`iso646.h`	Boolean Operators	ANSI
`limits.h`	Limits	ANSI
`locale.h`	Localization	ANSI
`math.h`	Mathematics	ANSI
`misra_types.h`	Exact-width integer types	MISRA-C:2004
`setjmp.h`	Non-Local Jumps	ANSI
`signal.h`	Signal Handling	ANSI
`stdarg.h`	Variable Arguments	ANSI
`stdbool.h`	Boolean macros	ANSI
`stddef.h`	Standard Definitions	ANSI
`stdfix.h`	Fixed point	ISO/IEC TR 18037
`stdint.h`	Exact width integer types	ANSI
`stdio.h`	Input/Output	ANSI

Table 1-10. Standard C Run-Time Library Header Files (Cont'd)

Header	Purpose	Standard
`stdlib.h`	Standard Library	ANSI
`string.h`	String Handling	ANSI
`time.h`	Date and Time	ANSI

The following sections provide descriptions of the header files contained in the C library. The header files are listed in alphabetical order.

adi_types.h

The `adi_types.h` header file contains the type definitions for `char_t`, `float32_t`, `float64_t`, and also includes both stdint.h and stdbool.h.

assert.h

The `assert.h` header file defines the `assert` macro, which can be used to insert run-time diagnostics into a source file. The macro normally tests (asserts) that an expression is true. If the expression is false, then the macro will first print an error message, and will then call the abort function to terminate the application. The message displayed by the `assert` macro will be of the form:

`ASSERT [expression]` fails at `"filename"`: `linenumber`

Note that the message includes the following information:

- `filename` - the name of the source file
- `linenumber` - the current line number in the source file
- `expression` - the expression tested

However if the macro `NDEBUG` is defined at the point at which the `assert.h` header file is included in the source file, then the assert macro will be defined as a null macro and no run-time diagnostics will be generated.

The strings associated with `assert.h` can be assigned to slower, more plentiful memory (and therefore free up faster memory) by placing a `default_section` pragma above the sections of code containing the asserts. For example:

```
#pragma default_section(STRINGS,"seg_sram")
```

Note that the pragma will affect the placement of all strings, and not just the ones associated with using the `ASSERT` macro. See the section "Linking Control Pragmas" in Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual* for more information about using the pragma.

An alternative to using the `default_section` pragma is to use the compiler's `-section` switch (for example `-section strings=seg_sram`). You can accomplish this in one of two ways:

- Use the command line.
- Use the **VisualDSP++ Project Options** dialog box. In the **Compile** category, select the **General** tab. Then type the command in the **Additional options:** field.

ctype.h

The `ctype.h` header file contains functions for character handling, such as `isalpha`, `tolower`, and so on.

For a list of library functions that use this header, see Table 1-19 on page 1-74.

cycle_count.h

The `cycle_count.h` header file provides an inexpensive method for benchmarking C-written source by defining basic facilities for measuring cycle counts. The facilities provided are based upon two macros, and a data type which are described in more detail in the section "Measuring Cycle Counts" on page 1-48.

cycles.h

The `cycles.h` header file defines a set of five macros and an associated data type that may be used to measure the cycle counts used by a section of C-written source. The macros can record how many times a particular piece of code has been executed and also the minimum, average, and maximum number of cycles used. The facilities that are available via this header file are described in the section "Measuring Cycle Counts" on page 1-48.

device.h

The `device.h` header file provides macros and defines data structures that an alternative device driver would require to provide file input and output services for `stdio` library functions. Normally, the `stdio` functions use a default driver to access an underlying device, but alternative device drivers may be registered that may then be used transparently by these functions. This mechanism is described in "Extending I/O Support To New Devices" on page 1-59.

device_int.h

The `device_int.h` header file contains function prototypes and provides enumerations for alterative device drivers. An alternative device driver is normally provided by an application and may be used by the `stdio` library functions to access an underlying device; an alternative device driver may coexist with, or may replace, the default driver that is supported by the VisualDSP++ simulator and EZ-KIT Lite® evaluation systems. Refer to "Extending I/O Support To New Devices" on page 1-59.

errno.h

The `errno.h` header file provides access to `errno` and also defines macros for associated error codes. This facility is not, in general, supported by the rest of the library.

float.h

The float.h header file defines the properties of the floating-point data types that are implemented by the compiler—that is, float, double, and long double. These properties are defined as macros and include the following for each supported data type:

- the maximum and minimum value (for example, FLT_MAX and FLT_MIN)
- the maximum and minimum power of ten (for example, FLT_MAX_10_EXP and FLT_MIN_10_EXP)
- the precision available expressed in terms of decimal digits (for example, FLT_DIG)
- a constant that represents the smallest value that may added to 1.0 and still result in a change of value (for example, FLT_EPSILON)

Note that the set of macros that define the properties of the double data type will have the same values as the corresponding set of macros for the float type when doubles are defined to be 32 bits wide, and they will have the same value as the macros for the long double data type when doubles are defined to be 64 bits wide (use the -double-size[-32|-64] compiler switch).

iso646.h

The iso646.h header file defines symbolic names for certain C operators; the symbolic names and their associated value are shown in Table 1-11.

Table 1-11. Symbolic Names Defined in iso646.h

Symbolic Name	Equivalent
`and`	&&
`and_eq`	&=
`bitand`	&
`bitor`	|
`compl`	~
`not`	!
`not_eq`	!=
`or`	||
`or_eq`	|=
`xor`	^
`xor_eq`	^=

ⓘ The symbolic names have the same name as the C++ keywords that are accepted by the compiler when the `-alttok` switch is specified.

limits.h

The `limits.h` header file contains definitions of maximum and minimum values for each C data type other than floating-point.

locale.h

The `locale.h` header file contains definitions for expressing numeric, monetary, time, and other data.

For a list of library functions that use this header, see Table 1-20 on page 1-74.

math.h

The `math.h` header file includes trigonometric, power, logarithmic, exponential, and other miscellaneous functions. The library contains the functions specified by the C standard along with implementations for the data types `float` and `long double`.

For a list of library functions that use this header, see Table 1-21 on page 1-75.

For every function that is defined to return a `double`, the `math.h` header file also defines corresponding functions that return a `float` and a `long double`. The names of the `float` functions are the same as the equivalent `double` function with `f` appended to its name. Similarly, the names of the `long double` functions are the same as the `double` function with `d` appended to its name.

For example, the header file contains the following prototypes for the `sine` function:

```
float sinf (float x);
double sin (double x);
long double sind (long double x);
```

When the compiler is treating `double` as 32 bits, the header file arranges that all references to the `double` functions are directed to the equivalent `float` function (with the suffix `f`). This allows you to use the un-suffixed names with arguments of type `double`, regardless of whether `doubles` are 32 or 64 bits long.

This header file also provides prototypes for a number of additional math functions provided by Analog Devices, such as `favg`, `fmax`, `fclip`, and `copysign`. Refer to Chapter 2, “DSP Run-Time Library” for more information about these additional functions.

The `math.h` header file also defines the macro `HUGE_VAL`. This macro evaluates to the maximum positive value that the type `double` can support.

The macros EDOM and ERANGE, defined in errno.h, are used by math.h functions to indicate domain and range errors.

A domain error occurs when an input argument is outside the domain of the function. "C Run-Time Library Reference" on page 1-79 lists the specific cases that cause errno to be set to EDOM, and the associated return values.

A range error occurs when the result of a function cannot be represented in the return type. If the result overflows, the function returns the value HUGE_VAL with the appropriate sign. If the result underflows, the function returns a zero without indicating a range error.

misra_types.h

The misra_types.h header file contains definitions of exact-width data types, as defined in "stdint.h" on page 1-28 and "stdbool.h" on page 1-27, plus data types char_t, float32_t and float64_t types.

setjmp.h

The setjmp.h header file contains setjmp and longjmp for non-local jumps.

For a list of library functions that use this header, see Table 1-22 on page 1-75.

signal.h

The signal.h header file provides function prototypes for the standard ANSI signal.h routines and also for several extensions, such as interrupt() and clear_interrupt().

The signal handling functions process conditions (hardware signals) that can occur during program execution. They determine the way that your C program responds to these signals. The functions are designed to process such signals as external interrupts and timer interrupts.

For a list of library functions that use this header, see Table 1-23 on page 1-75.

stdarg.h

The `stdarg.h` header file contains definitions needed for functions that accept a variable number of arguments. Programs that call such functions must include a prototype for the functions referenced.

For a list of library functions that use this header, see Table 1-24 on page 1-76.

stdbool.h

The `stdbool.h` header file contains three boolean related macros (`true`, `false` and `__bool_true_false_are_defined`) and an associated data type (`bool`). This header file was introduced in the C99 standard library.

stddef.h

The `stddef.h` header file contains a few common definitions useful for portable programs, such as `size_t`.

stdfix.h

The `stdfix.h` file contains function prototypes and macro definitions to support the native fixed-point type `fract` as defined by the ISO/IEC Technical Report 18037. The inclusion of this header file enables the `fract` keyword as an alias for `_Fract`. A discussion of support for native fixed-point types is given in "Using Native Fixed-Point Types" in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors*.

stdint.h

The `stdint.h` header file contains various exact-width integer types along with associated minimum and maximum values. The `stdint.h` header file was introduced in the C99 standard library.

Table 1-12 describes each type with regard to MIN and MAX macros.

Table 1-12. Exact-Width Integer Types

Type	Common Equivalent	MIN	MAX
int32_t	int	INT32_MIN	INT32_MAX
int64_t	long long	INT64_MIN	INT64_MAX
uint32_t	unsigned int	0	UINT32_MAX
uint64_t	unsigned long long	0	UINT64_MAX
int_least8_t	int	INT_LEAST8_MIN	INT_LEAST8_MAX
int_least16_t	int	INT_LEAST16_MIN	INT_LEAST16_MAX
int_least32_t	int	INT_LEAST32_MIN	INT_LEAST32_MAX
int_least64_t	long long	INT_LEAST64_MIN	INT_LEAST64_MAX
uint_least8_t	unsigned int	0	UINT_LEAST8_MAX
uint_least16_t	unsigned int	0	UINT_LEAST16_MAX
uint_least32_t	unsigned int	0	UINT_LEAST32_MAX
uint_least64_t	unsigned long long	0	UINT_LEAST64_MAX
int_fast8_t	int	INT_FAST8_MIN	INT_FAST8_MAX
int_fast16_t	int	INT_FAST16_MIN	INT_FAST16_MAX
int_fast32_t	int	INT_FAST32_MIN	INT_FAST32_MAX
int_fast64_t	long long	INT_FAST64_MIN	INT_FAST64_MAX
uint_fast8_t	unsigned int	0	UINT_FAST8_MAX
uint_fast16_t	unsigned int	0	UINT_FAST16_MAX
uint_fast32_t	unsigned int	0	UINT_FAST32_MAX
uint_fast64_t	unsigned int	0	UINT_FAST64_MAX

Table 1-12. Exact-Width Integer Types (Cont'd)

Type	Common Equivalent	MIN	MAX
intmax_t	int	INTMAX_MIN	INTMAX_MAX
intptr_t	int	INTPTR_MIN	INTPTR_MAX
uintmax_t	unsigned int	0	UINTMAX_MAX
uintptr_t	unsigned int	0	UINTPTR_MAX

Table 1-13 describes MIN and MAX macros defined for typedefs in other headings.

Table 1-13. MIN and MAX Macros for typedefs in Other Headings

Type	MIN	MAX
ptrdiff_t	PTRDIFF_MIN	PTRDIFF_MAX
sig_atomic_t	SIG_ATOMIC_MIN	SIG_ATOMIC_MAX
size_t	0	SIZE_MAX
wchar_t	WCHAR_MIN	WCHAR_MAX
wint_t	WINT_MIN	WINT_MAX

Macros for minimum-width integer constants include: INT8_C(x), INT16_C(x), INT32_C(x), UINT8_C(x), UINT16_C(x), UINT32_C(x), INT64_C(x) and UINT64_C(x).

Macros for greatest-width integer constants include INTMAX_C(x) and UINTMAX_C(x).

stdio.h

The `stdio.h` header file defines a set of functions, macros, and data types for performing input and output. Applications that use the facilities of this header file should link with the I/O library `libio.dlb` in the same way as linking with the C run-time library (see "Linking Library Functions" on page 1-4). The library is thread-safe but it is not interrupt-safe and should not therefore be called either directly or indirectly from an interrupt service routine.

The compiler uses definitions within the header file to select an appropriate set of functions that correspond to the currently selected size of type `double` (either 32 bits or 64 bits). Any source file that uses the facilities of `stdio.h` must therefore include the header file. Failure to include the header file results in a linker failure as the compiler must see a correct function prototype in order to generate the correct calling sequence.

The default I/O library does not support input and output of fixed-point values in floating-point format with the `r` and `R` format specifiers in the `printf` and `scanf` family of functions. These will be printed in hexadecimal format. If you wish to include full support for the `r` and `R` format specifiers, link your application with the fixed-point I/O library, using the `-flags-link -MD__LIBIO_FX` switch. For more information, see "Fixed-Point I/O Conversion Specifiers" in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors*.

The implementation of the `stdio.h` routines is based on a simple interface with a device driver that provides a set of low-level primitives for `open`, `close`, `read`, `write`, and `seek` operations. By default, these operations are provided by the VisualDSP++ simulator and EZ-KIT Lite systems and this mechanism is outlined in the section "Default Device Driver Interface" on page 1-68.

Alternative device drivers may be registered that can then be used transparently through the `stdio.h` functions. See "Extending I/O Support To New Devices" on page 1-59 for a description of the feature. Applications that do not require this functionality may be built with the `-flags-link -MD__LIBIO_LITE` switch. The switch links the application with a version of the I/O library that does not support the ability to register alternative device drivers, does not support the `%a` conversion specifier in `printf`, and does not support the `hh`, `j`, `ll`, `t`, or `z` size qualifiers in `scanf`. Linking with this switch results in a smaller executable.

When creating applications, be aware that the default device driver is activated when:

- A file is opened or closed.
- An input buffer becomes empty.
- An output buffer becomes full or is flushed.
- Interrogating or repositioning a file pointer.
- Deleting a file through the remove library function.
- Renaming a file through the rename library function.

Under the above conditions, the default device driver will disable interrupts and will halt the DSP while it negotiates with the host to perform the required I/O operation. Once the I/O operation has completed, the default device driver will restart the DSP and re-enable interrupts.

While the DSP is stopped, the cycle count registers are not updated and the DSP itself cannot initiate any interrupts; however, signals that correspond to external events can still occur, and these may be activated once the default device driver re-enables interrupts.

The following restrictions apply to this software release:

- The functions `tmpfile` and `tmpnam` are not available.
- The functions `rename` and `remove` are only supported under the default device driver supplied by the VisualDSP++ simulator and EZ-KIT Lite systems, and they only operate on the host file system.
- Positioning within a file that has been opened as a text stream is only supported if the lines within the file are terminated by the character sequence `\r\n`.
- Support for formatted reading and writing of data of `long double` type is only supported if an application is built with the `-double-size-64` switch.

At program termination, the host environment closes down any physical connection between the application and an opened file. However, the I/O library does not implicitly close any opened streams to avoid unnecessary overheads (particularly with respect to memory occupancy). Thus, unless explicit action is taken by an application, any unflushed output may be lost.

Any output generated by `printf` is always flushed but output generated by other library functions, such as `putchar`, `fwrite`, and `fprintf`, is not automatically flushed. Applications should therefore arrange to close down any streams that they open. Note that the function reference `fflush (NULL);` flushes the buffers of all opened streams.

> Each opened stream is allocated a buffer which either contains data from an input file or output from a program. For text streams, this data is held in the form of 8-bit characters that are packed into 32-bit memory locations. Due to internal mechanisms used to unpack and pack this data, the buffer must not reside at a memory location that is greater than the address `0x3fffffff`. Since the `stdio` library allocates buffers from the heap, this restriction implies that the heap should not be placed at address `0x40000000`

or above. The restriction may be avoided by using the `setvbuf` function to allocate the buffer from alternative memory, as in the following example.

```
#include <stdio.h>

char buffer[BUFSIZ];
setvbuf(stdout,buffer,_IOLBF,BUFSIZ);
printf("Hello World\n");
```

This example assumes that the buffer resides at a memory location that is less than `0x40000000`.

For a list of library functions that use this header, see Table 1-26 on page 1-76.

stdlib.h

The `stdlib.h` header file offers general utilities specified by the C standard. These include some integer math functions, such as `abs`, `div`, and `rand`; general string-to-numeric conversions; memory allocation functions, such as `malloc` and `free`; and termination functions, such as `exit`. This library also contains miscellaneous functions such as `bsearch` and `qsort`.

This header file also provides prototypes for a number of additional integer math functions provided by Analog Devices, such as `avg`, `max`, and `clip`. Table 1-14 is a summary of the additional library functions defined by the `stdlib.h` header file.

Some functions exist as both integer and floating point functions. The floating point functions typically have an `f` prefix. Make sure you use the correct type.

Table 1-14. Standard Library – Additional Functions

Description	Prototype

Average	`int avg (int a, int b);`
	`long lavg (long a, long b);`
	`long long llavg (long long a, long long b);`

Clip	`int clip (int a, int b);`
	`long lclip (long a, long b);`
	`long long llclip (long long a, long long b);`

Count bits set	`int count_ones (int a);`
	`int lcount_ones (long a);`
	`int llcount_ones (long long a);`

Maximum	`int max (int a, int b);`
	`long lmax (long a, long b);`
	`long long llmax (long long a, long long b);`

Minimum	`int min (int a, int b);`
	`long lmin (long a, long b);`
	`long long llmin (long long a, long long b);`

Multiple heaps for dynamic memory allocation	`void *heap_calloc(int heap_index, size_t nelem, size_t size);`
	`void heap_free(int heap_index, void *ptr);`
	`void *heap_malloc(int heap_index, size_t size);`
	`void *heap_realloc(int heap_index, void *ptr, size_t size);`
	`int set_alloc_type(char * heap_name);`
	`int heap_install(void *base, size_t size, int userid, int pmdm);`
	`int heap_lookup_name(char *userid);`
	`int heap_switch(int heapid);`

A number of functions, including `abs`, `avg`, `max`, `min`, and `clip`, are implemented via intrinsics (provided the header file has been `#include`'d) that map to single-cycle machine instructions.

If the header file is not included, the library implementation is used instead—at a considerable loss in efficiency.

For a list of library functions that use this header, see Table 1-27 on page 1-77.

string.h

The `string.h` header file contains string handling functions, including `strcpy` and `memcpy`.

For a list of library functions that use this header, see Table 1-28 on page 1-78.

time.h

The `time.h` header file provides functions, data types, and a macro for expressing and manipulating date and time information. The header file defines two fundamental data types, one of which is `clock_t` and the other which is `time_t`.

The `time_t` data type is used for values that represent the number of seconds that have elapsed since a known epoch; values of this form are known as a *calendar time*. In this implementation, the epoch starts on 1st January, 1970, and calendar times before this date are represented as negative values.

A calendar time may also be represented in a more versatile way as a broken-down time which is a structured variable of the following form:

```
struct tm { int tm_sec;  /* seconds after the minute [0,61] */
  int tm_min;           /* minutes after the hour [0,59]   */
  int tm_hour;          /* hours after midnight [0,23]     */
  int tm_mday;          /* day of the month [1,31]         */
  int tm_mon;           /* months since January [0,11]     */
  int tm_year;          /* years since 1900                */
  int tm_wday;          /* days since Sunday [0, 6]        */
  int tm_yday;          /* days since January 1st [0,365]  */
  int tm_isdst;         /* Daylight Saving flag            */
};
```

This implementation does not support either the Daylight Saving flag in the structure struct tm; nor does it support the concept of time zones. All calendar times are therefore assumed to relate to Greenwich Mean Time (Coordinated Universal Time or UTC).

The `clock_t` data type is associated with the number of implementation-dependent processor “ticks” used since an arbitrary starting point. By default the data type is equivalent to the `long` data type and can only be used to measure an elapsed time of a small number of seconds (depending upon the processor’s clock speed). To measure a longer time span requires an alternative definition of the data type.

If the macro `__LONG_LONG_PROCESSOR_TIME__` is defined at compile-time (either before including the header file `time.h`, or by using the compile-time switch `-D__LONG_LONG_PROCESSOR_TIME__`), the `clock_t` data type will be `typedef`’d as a `long long`, which should be sufficient to record an elapsed time for the most demanding application.

The header file sets the `CLOCKS_PER_SEC` macro to the number of processor cycles per second and this macro can therefore be used to convert data of type `clock_t` into seconds, normally by using floating-point arithmetic to divide it into the result returned by the `clock` function.

In general, the processor speed is a property of a particular chip and it is therefore recommended that the value to which this macro is set is verified independently before it is used by an application.

In this version of the C/C++ compiler, the `CLOCKS_PER_SEC` macro is set by one of the following (in descending order of precedence):

- Via the `-DCLOCKS_PER_SEC=<definition>` compile-time switch
- Via the **Processor speed** box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Processor** category
- From the header file `cycles.h`

For a list of library functions that use this header, see Table 1-29 on page 1-78.

Calling Library Functions From an ISR

Not all C run-time library functions are interrupt-safe (and can therefore be called from an interrupt service routine). For a run-time function to be classified as *interrupt-safe*, it must:

- Not update any global data, such as `errno`, and
- Not write to (or maintain) any private static data

It is recommended therefore that none of the functions defined in the header file `math.h`, nor the string conversion functions defined in `stdlib.h`, be called from an ISR as these functions are commonly defined to update the global variable `errno`. Similarly, the functions defined in the `stdio.h` header file maintain static tables for currently opened streams and should not be called from an ISR. Additionally, the memory allocation routines malloc, calloc, realloc, free, and the C++ operators new and delete read and update global tables and are not interrupt-safe.

Several other library functions are not interrupt-safe because they make use of private static data. These functions are:

```
asctime
gmtime
localtime
rand
srand
strtok
```

While not all C run-time library functions are interrupt-safe, versions of the functions are available that are *thread-safe* and may be used in a VDK multi-threaded environment. These library functions can be found in the run-time libraries that have the suffix `_mt` in their filename.

Using the Libraries in a Multi-Threaded Environment

It is sometimes desirable for there to be several instances of a given library function to be active at any one time. Two examples of such a requirement are:

- An interrupt or other external event invokes a function, while the application is also executing that function,
- An application that runs in a multi-threaded environment, such as VDK, and more than one thread executes the function concurrently.

The majority of the functions in the C and C++ run-time libraries are safe in this regard and may be called in either of the above schemes; this is because the functions operate on parameters passed in by the caller and they do not maintain private static storage, and they do not access non-constant global data.

A subset of the library functions however either make use of private storage or they operate on shared resources (such as `FILE` pointers). This can lead to undefined behavior if two instances of a function simultaneously access the same data. The issues associated with calling such library functions via an interrupt or other external event is discussed in the section "Calling Library Functions From an ISR" on page 1-37.

A VisualDSP++ installation contains versions of the C and C++ libraries that may be used in a multi-threaded environment. These libraries have recursive locking mechanisms so that shared resources, such as `stdio FILE` tables and buffers, are only updated by a single function instance at any given time. The libraries also make use of local-storage routines for thread-local private copies of data, and for the variable `errno` (each thread therefore has its own copy of `errno`).

The multi-threaded libraries have "mt" in their filename and will be used automatically by the default VDK .ldf file to build a multi-threaded application.

Note that the DSP run-time library (which is described in Chapter 2, "DSP Run-Time Library") is thread-safe and may be used in any multi-threaded environment.

Using Compiler Built-In C Library Functions

The C compiler intrinsic (built-in) functions are functions that the compiler immediately recognizes and replaces with inline assembly code instead of a function call. For example, the absolute value function, `abs()`, is recognized by the compiler, which subsequently replaces a call to the C run-time library version with an inline version. The `cc21k` compiler contains a number of intrinsic built-in functions for efficient access to various features of the hardware.

Built-in functions are recognized for cases where the name begins with the string `__builtin`, and the declared prototype of the function matches the prototype that the compiler expects. Built-in functions are declared in system header files. Include the appropriate header file in your program to use these functions. The normal action of the appropriate include file is to `#define` the normal name as mapping to the built-in form.

Typically, inline built-in functions are faster than an average library routine, and it does not incur the calling overhead. The routines in Table 1-15 are built-in C library functions for the `cc21k` compiler.

Table 1-15. Compiler Built-in Functions

abs	avg	clip
copysign[1]	copysignf	fabs[1]
fabsf	favg[1]	favgf
fclip[1]	fclipf	fmax[1]
fmaxf	fmin[1]	fminf
labs	lavg	lclip
lmax	lmin	max
min		

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

If you want to use the C run-time library functions of the same name, compile with the `-no-builtin` compiler switch.

For a certain category of library function, the compiler relaxes the normal rule whereby pointers that are passed as arguments must address Data Memory (DM). For functions in this category, any argument that is a pointer may also address Program Memory (PM). When the compiler recognizes that certain arguments reference PM, it generates a call to an appropriate version of the function in the run-time library.

Table 1-16 contains a list of library functions that may be called with pointers to Program Memory. Note that this facility is only available provided that the `-no-builtin` compiler switch has not been specified.

Table 1-16. Dual Memory Capable Functions

atof	atoi	atol	frexp
frexpf	memchr	memcmp	memcpy
memmove	memset	modf	modff
setlocale	strcat	strchr	strcmp
strcoll	strcpy	strcspn	strlen
strncat	strncmp	strncpy	strpbrk
strrchr	strspn	strstr	strtod
strtok	strtol	strtoul	strxfrm

Abridged C++ Library Support

When in C++ mode, the `cc21k` compiler can call a large number of functions from the Abridged Library, a conforming subset of C++ library.

C++ is not supported for ADSP-21020 processors.

The Abridged C++ library has two major components: embedded C++ library (EC++) and embedded standard template library (ESTL). The embedded C++ library is a conforming implementation of the embedded C++ library as specified by the Embedded C++ Technical Committee. You can view the Abridged Library Reference by locating the file `docs\cpl_lib\index.html` underneath your `VisualDSP++` installation and opening it in a web browser.

This section lists and briefly describes the following components of the Abridged C++ library:

- "Embedded C++ Library Header Files" on page 1-42
- "C++ Header Files for C Library Facilities" on page 1-44

- "Embedded Standard Template Library Header Files" on page 1-46
- "Using Thread-Safe C/C++ Run-Time Libraries With VDK" on page 1-48

For more information on the Abridged Library, see online Help.

Embedded C++ Library Header Files

The following section provides a brief description of the header files in the embedded C++ library.

complex

The `complex` header file defines a template class `complex` and a set of associated arithmetic operators. Predefined types include `complex_float` and `complex_long_double`.

This implementation does not support the full set of complex operations as specified by the C++ standard. In particular, it does not support either the transcendental functions or the I/O operators << and >>.

The `complex` header and the C library header file `complex.h` refer to two different and incompatible implementations of the `complex` data type.

exception

The `exception` header file defines the `exception` and `bad_exception` classes and several functions for exception handling.

fract

The `fract` header file defines the `fract` data type, which supports fractional arithmetic, assignment, and type-conversion operations. The header file is fully described in Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, section "C++ Fractional Type Support".

fstream

The `fstream` header file defines the `filebuf`, `ifstream`, and `ofstream` classes for external file manipulations.

iomanip

The `iomanip` header file declares several `iostream` manipulators. Each manipulator accepts a single argument.

ios

The `ios` header file defines several classes and functions for basic `iostream` manipulations. Note that most of the `iostream` header files include `ios`.

iosfwd

The `iosfwd` header file declares forward references to various `iostream` template classes defined in other standard header files.

iostream

The `iostream` header file declares most of the `iostream` objects used for the standard stream manipulations.

istream

The `istream` header file defines the `istream` class for `iostream` extractions. Note that most of the `iostream` header files include `istream`.

new

The `new` header file declares several classes and functions for memory allocations and deallocations.

ostream

The `ostream` header file defines the `ostream` class for `iostream` insertions.

sstream

The sstream header file defines the stringbuf, istringstream, and ostringstream classes for various string object manipulations.

stdexcept

The stdexcept header file defines a variety of classes for exception reporting.

streambuf

The streambuf header file defines the streambuf classes for basic operations of the iostream classes. Note that most of the iostream header files include streambuf.

string

The string header file defines the string template and various supporting classes and functions for string manipulations.

Objects of the string type should not be confused with the null-terminated C strings.

strstream

The strstream header file defines the strstreambuf, istrstream, and ostream classes for iostream manipulations on allocated, extended, and freed character sequences.

C++ Header Files for C Library Facilities

For each C standard library header there is a corresponding standard C++ header. If the name of a C standard library header file were foo.h, then the name of the equivalent C++ header file would be cfoo. For example, the C++ header file <cstdio> provides the same facilities as the C header file <stdio.h>.

Table 1-17 lists the C++ header files that provide access to the C library facilities.

The C standard headers files may be used to define names in the C++ global namespace, while the equivalent C++ header files define names in the standard namespace.

Table 1-17. C++ Header Files for C Library Facilities

Header	Description
`cassert`	Enforces assertions during function executions
`cctype`	Classifies characters
`cerrno`	Tests error codes reported by library functions
`cfloat`	Tests floating-point type properties
`climits`	Tests integer type properties
`clocale`	Adapts to different cultural conventions
`cmath`	Provides common mathematical operations
`csetjmp`	Executes non-local goto statements
`csignal`	Controls various exceptional conditions
`cstdarg`	Accesses a variable number of arguments
`cstddef`	Defines several useful data types and macros
`cstdio`	Performs input and output
`cstdlib`	Performs a variety of operations
`cstring`	Manipulates several kinds of strings

Chapter 2, “DSP Run-Time Library” describes the functions in the DSP run-time libraries. Referencing these functions with a namespace prefix is not supported. All DSP library functions are in the global namespace.

Embedded Standard Template Library Header Files

Templates and the associated header files are not part of the embedded C++ standard, but they are supported by the `cc21k` compiler in C++ mode.

The embedded standard template library header files are:

algorithm

The `algorithm` header file defines numerous common operations on sequences.

deque

The `deque` header file defines a deque template container.

functional

The `functional` header file defines numerous function templates that can be used to create callable types.

hash_map

The `hash_map` header file defines two hashed map template containers.

hash_set

The `hash_set` header file defines two hashed set template containers.

iterator

The `iterator` header file defines common iterators and operations on iterators.

list

The `list` header file defines a list template container.

map

The `map` header file defines two map template containers.

memory

The `memory` header file defines facilities for managing memory.

numeric

The `numeric` header file defines several numeric operations on sequences.

queue

The `queue` header file defines two queue template container adapters.

set

The `set` header file defines two set template containers.

stack

The `stack` header file defines a stack template container adapter.

utility

The `utility` header file defines an assortment of utility templates.

vector

The `vector` header file defines a vector template container.

Header Files for C++ Library Compatibility

The Embedded C++ library also includes several header files for compatibility with traditional C++ libraries. Table 1-18 describes these files.

Table 1-18. Header Files for C++ Library Compatibility

Header	Description
`fstream.h`	Defines several `iostream` template classes that manipulate external files
`iomanip.h`	Declares several iostreams manipulators that take a single argument
`iostream.h`	Declares the `iostream` objects that manipulate the standard streams
`new.h`	Declares several functions that allocate and free storage

Using Thread-Safe C/C++ Run-Time Libraries With VDK

When developing for VDK, the thread-safe variants of the run-time libraries are linked with user applications. These libraries may add an overhead to the VDK resources required by some applications.

The run-time libraries make use of VDK synchronicity functions to ensure thread safety.

Measuring Cycle Counts

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

Each of these methods is described in:

- "Basic Cycle Counting Facility" on page 1-49
- "Cycle Counting Facility With Statistics" on page 1-51
- "Using time.h to Measure Cycle Counts" on page 1-54
- "Determining the Processor Clock Rate" on page 1-56
- "Considerations When Measuring Cycle Counts" on page 1-57

Basic Cycle Counting Facility

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

```
START_CYCLE_COUNT(S)
```

```
STOP_CYCLE_COUNT(T,S)
```

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

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

The `cycle_t` type can be configured to use the unsigned long long type for its definition. To do this, you should compile your application with the compile-time macro `__LONG_LONG_PROCESSOR_TIME__` defined to 1.

The header file also defines the macro:

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

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

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

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

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

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

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

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

The run-time libraries provide alternative facilities for measuring the performance of C source (see "Cycle Counting Facility With Statistics" on page 1-51 and "Using time.h to Measure Cycle Counts" on page 1-54); the relative benefits of this facility are outlined in "Considerations When Measuring Cycle Counts" on page 1-57.

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

Cycle Counting Facility With Statistics

The `cycles.h` header file defines a set of macros for measuring the performance of compiled C source. In addition to providing the basic facility for reading the `EMUCLK` cycle count register of the SHARC architecture, the macros can also accumulate statistics suited to recording the performance of a section of code that is executed repeatedly.

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

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

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

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

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

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

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

```
AVG   : 95
MIN   : 92
MAX   : 100
CALLS : 4
```

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

```
CYCLES : 95
```

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

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

The following example demonstrates how this facility may be used.

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

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

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

   CYCLES_INIT(stats);

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

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

This example might output:

```
Cycles used by foo
   AVG   : 25454
   MIN   : 23003
   MAX   : 26295
   CALLS : 16

Cycles used by bar
   AVG   : 8727
   MIN   : 7653
   MAX   : 8912
   CALLS : 16
```

Alterative methods of measuring the performance of compiled C source are described in the sections "Basic Cycle Counting Facility" on page 1-49 and "Using time.h to Measure Cycle Counts" on page 1-54. Also refer to "Considerations When Measuring Cycle Counts" on page 1-57 which provides some useful tips with regards to performance measurements.

Using time.h to Measure Cycle Counts

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

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

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

If this method of timing is used by an application, note that:

- The value assigned to the macro `CLOCKS_PER_SEC` should be independently verified to ensure that it is correct for the particular processor being used (see "Determining the Processor Clock Rate" on page 1-56),
- The result returned by the `clock` function does not include the overhead of calling the library function.

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

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

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

    double secs;

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

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

The `cycles.h` and `cycle_count.h` header files define other methods for benchmarking an application—these header files are described in the sections "Basic Cycle Counting Facility" on page 1-49 and "Cycle Counting Facility With Statistics" on page 1-51, respectively. Also refer to "Considerations When Measuring Cycle Counts" on page 1-57 which provides some guidelines that may be useful.

Determining the Processor Clock Rate

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

Measuring the amount of time that an application takes to run on a SHARC processor usually involves first determining the number of cycles that the processor takes, and then dividing this value by the processor's clock rate. The `time.h` header file defines the macro `CLOCKS_PER_SEC` as the number of processor "ticks" per second.

On an ADSP-21xxx (SHARC) architecture, this parameter is set by the run-time library to one of the following values in descending order of precedence:

- By way of the compile-time switch `-DCLOCKS_PER_SEC=<definition>`.
- By way of the **Processor speed** box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Processor** category
- From the `cycles.h` header file

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

Considerations When Measuring Cycle Counts

This section summarizes cycle-counting techniques for benchmarking C-compiled code. Each of these alternatives are described below.

- "Basic Cycle Counting Facility" on page 1-49
  The basic cycle counting facility represents an inexpensive and relatively unobtrusive method for benchmarking C-written source using cycle counts. The facility is based on macros that factor in the overhead incurred by the instrumentation. The macros may be customized and can be switched either or off, and so no source changes are required when moving between development and release mode. The same set of macros is available on other platforms provided by Analog Devices.
- "Cycle Counting Facility With Statistics" on page 1-51
  This cycle-counting facility has more features than the basic cycle counting facility described above. It is more expensive in terms of program memory, data memory, and cycles consumed. However, it can record the number of times that the instrumented code has been executed and can calculate the maximum, minimum, and average cost of each iteration. The provided macros take into account the overhead involved in reading the cycle count register. By default, the macros are switched off, but they can be switched on by specifying the `-DDO_CYCLE_COUNTS` compile-time switch. The macros may be customized for a specific application. This cycle counting facility is also available on other Analog Devices architectures.

- "Using time.h to Measure Cycle Counts" on page 1-54
  The facilities of the `time.h` header file represent a simple method for measuring the performance of an application that is portable across many different architectures and systems. These facilities are based on the `clock` function.

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

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

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

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

The different methods presented here to obtain the performance metrics of an application are based on the `EMUCLK` register. This is a 32-bit register that is incremented at every processor cycle; once the counter reaches the value `0xffffffff` it will wrap back to zero and will also increment the `EMUCLK2` register. By default, to save memory and execution time, the `EMUCLK2` register is not used by either the `clock` function or the cycle counting macros. The performance metrics therefore will wrap back to zero after approximately every 71 seconds on a 60 MHz processor. If you require a longer measurement duration, define the compile-time macro `__LONG_LONG_PROCESSOR_TIME__`.

File I/O Support

The VisualDSP++ environment provides access to files on a host system by using stdio functions. File I/O support is provided through a set of low-level primitives that implement the open, close, read, write, and seek operations. The functions defined in the `stdio.h` header file make use of these primitives to provide conventional C input and output facilities. The source files for the I/O primitives are available under the ADSP-21xxx installation of VisualDSP++ in the subdirectory `...\lib\src\libio_src`.

This section describes:

- "Extending I/O Support To New Devices" on page 1-59
- "Default Device Driver Interface" on page 1-68

Refer to "stdio.h" on page 1-30 for information about the conventional C input and output facilities that are provided by the compiler.

Extending I/O Support To New Devices

The I/O primitives are implemented using an extensible device driver mechanism. The default start-up code includes a device driver that can perform I/O through the VisualDSP++ simulator and EZ-KIT Lite evaluation systems. Other device drivers may be registered and then used through the normal `stdio` functions.

This section describes:

- "DevEntry Structure" on page 1-60
- "Registering New Devices" on page 1-65
- "Pre-Registering Devices" on page 1-66
- "Default Device" on page 1-67
- "Remove and Rename Functions" on page 1-68

DevEntry Structure

A device driver is a set of primitive functions grouped together into a `DevEntry` structure. This structure is defined in `device.h`.

```
struct DevEntry {
   int   DeviceID;
   void  *data;

   int   (*init)(struct DevEntry *entry);
   int   (*open)(const char *name, int mode);
   int   (*close)(int fd);
   int   (*write)(int fd, unsigned char *buf, int size);
   int   (*read)(int fd, unsigned char *buf, int size);
   long  (*seek)(long fd, int offset, int whence);
   int stdinfd;
   int stdoutfd;
   int stderrfd;
}

typedef struct DevEntry DevEntry;
typedef struct DevEntry *DevEntry_t;
```

The fields within the `DevEntry` structure have the following meanings.

DeviceID:
The `DeviceID` field is a unique identifier for the device, known to the user. Device IDs are used globally across an application.

data:
The `data` field is a pointer for any private data the device may need; it is not used by the run-time libraries.

init:
The `init` field is a pointer to an initialization function. The run-time library calls this function when the device is first registered, passing in the address of this structure (and thus giving the `init` function access to `DeviceID` and the field `data`). If the `init` function encounters an error, it must return -1. Otherwise, it must return a positive value to indicate success.

open:
The `open` field is a pointer to a function performs the "*`open file`*" operation upon the device; the run-time library will call this function in response to requests such as `fopen()`, when the device is the currently-selected default device. The `name` parameter is the path name to the file to be opened, and the `mode` parameter is a bitmask that indicates how the file is to be opened:

```
0x0001   Open file for reading
0x0002   Open file for writing
0x0004   Open file for appending
0x0008   Truncate the file to zero length, if it already exists
0x00010  Create the file, if it does not already exist
```

By default, files are opened as text streams (in which the character sequence `\r\n` is converted to `\n` when reading, and the character `\n` is written to the file as `\r\n`). A file is opened as a binary stream if the following bit value is set in the `mode` parameter:

`0x0020   Open the file as a binary stream (raw mode).`

The `open` function must return a positive "*`file descriptor`*" if it succeeds in opening the file; this file descriptor is used to identify the file to the device in subsequent operations. The file descriptor must be unique for all files currently open for the device, but need not be distinct from file descriptors returned by other devices—the run-time library identifies the file by the combination of device and file descriptor.

If the `open` function fails, it must return -1 to indicate failure.

close:
The `close` field is a pointer to a function that performs the "*`close file`*" operation on the device. The run-time library calls the `close` function in response to requests such as `fclose()` on a stream that was opened on the device. The `fd` parameter is a file descriptor previously returned by a call to the `open` function. The `close` function must return a zero value for success, and a non-zero value for failure.

write:
The `write` field is a pointer to a function that performs the "*`write to file`*" operation on the device. The run-time library calls the `write` function in response to requests, such as `fwrite()`, `fprintf()` and so on, that act on streams that were opened on the device. The `write` function takes three parameters:

- `fd` – this is a file descriptor that identifies the file to be written to; it will be a value that was returned from a previous call to the open function.

- `buf` – a pointer to the data to be written to the file
- `size` – the number of bytes to be written to the file

The `write` function must return one of the following values:

- A positive value from 1 to `size` inclusive, indicating how many bytes from `buf` were successfully written to the file
- Zero, indicating that the file has been closed, for some reason (for example, network connection dropped)
- A negative value, indicating an error

read:
The `read` field is a pointer to a function that performs the "*`read from file`*" operation on the device. The run-time library calls the `read` function in response to requests, such as `fread()`, `fscanf()` and so on, that act on streams that were opened on the device. The `read` function's parameters are:

- `fd` – this is the file descriptor for the file to be read
- `buf` – this is a pointer to the buffer where the retrieved data must be stored
- `size` – this is the number of (8-bit) bytes to read from the file. This must not exceed the space available in the buffer pointed to by `buf`

The `read` function must return one of the following values:

- A positive value from 1 to `size` inclusive, indicating how many bytes were read from the file into `buf`
- Zero, indicating end-of-file
- A negative value, indicating an error

> ⓘ The run-time library expects the `read` function to return `0xa (10)` as the newline character.

seek:
The `seek` field is a pointer to a function that performs dynamic access on the file. The run-time library calls the `seek` function in response to requests such as `rewind()`, `fseek()`, and so on, that act on streams that were opened on the device.

The `seek` function takes the following parameters:

- `fd` – this is the file descriptor for the file which will have its read/write position altered
- `offset` – this is a value that is used to determine the new read/write pointer position within the file; it is in (8-bit) bytes
- `whence` – this is a value that indicates how the `offset` parameter is interpreted:
  - 0: `offset` is an absolute value, giving the new read/write position in the file
  - 1: `offset` is a value relative to the current position within the file
  - 2: `offset` is a value relative to the end of the file

The `seek` function returns a positive value that is the new (absolute) position of the read/write pointer within the file, unless an error is encountered, in which case the `seek` function must return a negative value.

If a device does not support the functionality required by one of these functions (such as read-only devices, or stream devices that do not support seeking), the `DevEntry` structure must still have a pointer to a valid function; the function must arrange to return an error for attempted operations.

stdinfd:
The `stdinfd` field is set to the device file descriptor for `stdin` if the device is expecting to claim the `stdin` stream, or to the enumeration value `dev_not_claimed` otherwise.

stdoutfd:
The `stdoutfd` field is set to the device file descriptor for `stdout` if the device is expecting to claim the `stdout` stream, or to the enumeration value `dev_not_claimed` otherwise.

stderrfd:
The `stderrfd` field is set to the device file descriptor for `stderr` if the device is expecting to claim the `stderr` stream, or to the enumeration value `dev_not_claimed` otherwise.

Registering New Devices

A new device can be registered with the following function:

```
int add_devtab_entry(DevEntry_t entry);
```

If the device is successfully registered, the `init()` routine of the device is called, with `entry` as its parameter. The `add_devtab_entry()` function returns the `DeviceID` of the device registered.

If the device is not successfully registered, a negative value is returned. Reasons for failure include (but are not limited to):

- The `DeviceID` is the same as another device, already registered
- There are no more slots left in the device registry table
- The `DeviceID` is less than zero
- Some of the function pointers are NULL

- The device's `init()` routine returned a failure result
- The device has attempted to claim a standard stream that is already claimed by another device

Pre-Registering Devices

The library source file `devtab.c` (which can be found under a VisualDSP++ installation in the subdirectory `...\lib\src\libio_src`) declares the array:

```
DevEntry_t DevDrvTable[];
```

This array contains pointers to `DevEntry` structures for each device that is pre-registered, that is, devices that are available as soon as `main()` is entered, and that do not need to be registered at run-time by calling `add_devtab_entry()`. By default, the "*PrimIO*" device is registered. The `PrimIO` device provides support for target/host communication when using the simulators and the Analog Devices emulators and debug agents. This device is pre-registered, so that `printf()` and similar functions operate as expected without additional setup.

Additional devices can be pre-registered by the following process:

1. Take a copy of the `devtab.c` source file and add it to your project.
2. Declare your new device's `DevEntry` structure within the `devtab.c` file, for example,

   ```
   extern DevEntry myDevice;
   ```

3. Include the address of the `DevEntry` structure within the `DevDrvTable[]` array. Ensure that the table is null-terminated. For example,

```
DevEntry_t DevDrvTable[MAXDEV] = {
#ifdef PRIMIO
   &primio_deventry,
```

```
#endif
   &myDevice,   /* new pre-registered device */
   0,
};
```

All pre-registered devices are initialized by the run-time library when it calls the `init` function of each of the pre-registered devices in turn.

The normal behavior of the `PrimIO` device when it is registered is to claim the first three files as `stdin`, `stdout` and `stderr`. These standard streams may be re-opened on other devices at run-time by using `freopen()` to close the `PrimIO`-based streams and re-open the streams on the current default device.

To allow an alternative device (either pre-registered or registered by `add_devtab_entry()`) to claim one or all of the standard streams:

1. Take a copy of the `primiolib.c` source file, and add it to your project.
2. Edit the appropriate `stdinfd`, `stdoutfd`, and `stderrfd` file descriptors in the `primio_deventry` structure to have the value `dev_not_claimed`.
3. Ensure the alternative device's `DevEntry` structure has set the standard stream file descriptors appropriately.

Both the device initialization routines, called from the startup code and `add_devtab_entry()`, return with an error if a device attempts to claim a standard stream that is already claimed.

Default Device

Once a device is registered, it can be made the default device using the following function:

```
void set_default_io_device(int);
```

The function should be passed the `DeviceID` of the device. There is a corresponding function for retrieving the current default device:

```
int get_default_io_device(void);
```

The default device is used by `fopen()` when a file is first opened. The `fopen()` function passes the open request to the `open()` function of the device indicated by `get_default_io_device()`. The device's file identifier (`fd`) returned by the `open()` function is private to the device; other devices may simultaneously have other open files that use the same identifier. An open file is uniquely identified by the combination of `DeviceID` and `fd`.

The `fopen()` function records the `DeviceID` and `fd` in the global open file table, and allocates its own internal `fid` to this combination. All future operations on the file use this `fid` to retrieve the `DeviceID` and thus direct the request to the appropriate device's primitive functions, passing the `fd` along with other parameters. Once a file has been opened by `fopen()`, the current value of `get_default_io_device()` is irrelevant to that file.

Remove and Rename Functions

The `PrimIO` device provides support for the `remove()` and `rename()` functions. These functions are not currently part of the extensible File I/O interface, since they deal purely with path names, and not with file descriptors. All calls to `remove()` and `rename()` in the run-time library are passed directly to the `PrimIO` device.

Default Device Driver Interface

The stdio functions provide access to the files on a host system through a device driver that supports a set of low-level I/O primitives. These low-level primitives are described under "Extending I/O Support To New Devices" on page 1-59. The default device driver implements these primitives based on a simple interface provided by the VisualDSP++ simulator and EZ-KIT Lite systems.

All the I/O requests submitted through the default device driver are channeled through the C function `_primIO`. The assembly label has two underscores, `__primIO` . The source for this function, and all the other library routines, can be found under the base installation for VisualDSP++ in the subdirectory `...\lib\src\libio_src`.

The `__primIO` function accepts no arguments. Instead, it examines the I/O control block at the label `_PrimIOCB`. Without external intervention by a host environment, the `__primIO` routine simply returns, which indicates failure of the request. Two schemes for host interception of I/O requests are provided.

The first scheme is to modify control flow into and out of the `__primIO` routine. Typically, this would be achieved by a break point mechanism available to a debugger/simulator. Upon entry to `__primIO`, the data for the request resides in a control block at the label `_PrimIOCB`. If this scheme is used, the host should arrange to intercept control when it enters the `__primIO` routine, and, after servicing the request, return control to the calling routine.

The second scheme involves communicating with the DSP processor through a pair of simple semaphores. This scheme is most suitable for an externally-hosted development board. Under this scheme, the host system should clear the data word whose label is `__lone_SHARC`; this causes `__primIO` to assume that a host environment is present and able to communicate with the process.

If `__primIO` sees that `__lone_SHARC` is cleared, then upon entry (for example, when an I/O request is made) it sets a non-zero value into the word labeled `__Godot`. The `__primIO` routine then busy-waits until this word is reset to zero by the host. The non-zero value of `__Godot` raised by `__primIO` is the address of the I/O control block.

Data Packing for Primitive I/O

The implementation of the __primIO interface is based on a word-addressable machine, with each word comprising a fixed number of 8-bit bytes. All READ and WRITE requests specify a move of some number of 8-bit bytes, that is, the relevant fields count 8-bit bytes, not words. Packing is always little endian, the first byte of a file read or written is the low-order byte of the first word transferred.

Data packing is set to four bytes per word for the SHARC architecture. Data packing can be changed to accommodate other architectures by modifying the constant BITS_PER_WORD, defined in _wordsize.h. (For example, a processor with 16-bit addressable words would change this value to 16).

Note that the file name provided in an OPEN request uses the processor's "native" string format, normally one byte per word. Data packing applies only to READ and WRITE requests.

Data Structure for Primitive I/O

The I/O control block is declared in _primio.h, as follows.

```
typedef struct
{
   enum
   {
      PRIM_OPEN = 100,
      PRIM_READ,
      PRIM_WRITE,
      PRIM_CLOSE,
      PRIM_SEEK,
      PRIM_REMOVE,
      PRIM_RENAME
   } op;
   int   fileID;
```

```
    int   flags;
    unsigned char *buf;   /* data buffer, or file name     */
    int   nDesired;       /* number of characters to read  */
                          /* or write                      */
    int   nCompleted;     /* number of characters actually */
                          /* read or written               */
    void  *more;          /* for future use                */
}
PrimIOCB_T;
```

The first field, `op`, identifies which of the seven currently-supported operations is being requested.

The file ID for an open file is a non-negative integer assigned by the debugger or other "host" mechanism. The `fileID` values 0, 1, and 2 are pre-assigned to `stdin`, `stdout`, and `stderr`, respectively. No open request is required for these file IDs.

Before "activating" the debugger or other host environment, an OPEN or REMOVE request may set the `fileID` field to the length of the filename to open or delete; a RENAME request may also set the field to the length of the old filename. If the `fileID` field does contain a string length, then this will be indicated in the `flags` field (see below), and the debugger or other host environment will be able to use the information to perform a batch memory read to extract the filename. If the information is not provided, then the file name has to be extracted one character at a time.

The `flags` field is a bit-field containing other information for special requests. Meaningful bit values for an `OPEN` operation are:

```
M_OPENR = 0x0001     /* open for reading                  */
M_OPENW = 0x0002     /* open for writing                  */
M_OPENA = 0x0004     /* open for append                   */
M_TRUNCATE = 0x0008  /* truncate to zero length if file exists */
M_CREATE = 0x0010    /* create the file if necessary      */
```

```
M_BINARY = 0x0020    /* binary file (vs. text file)        */
M_STRLEN_PROVIDED = 0x8000 /* length of file name(s) available */
```

For a `READ` operation, the low-order four bits of the `flag` value contain the number of bytes packed into each word of the read buffer, and the rest of the value is reserved for future use.

For a `WRITE` operation, the low-order four bits of the `flag` value contain the number of bytes packed into each word of the write buffer, and the rest of the value form a bit-field, for which only the following bit is currently defined:

```
M_ALIGN_BUFFER = 0x10
```

If this bit is set for a `WRITE` request, the `WRITE` operation is expected to be aligned on a processor word boundary by writing padding NULLs to the file before the buffer contents are transferred.

For an `OPEN`, `REMOVE`, and `RENAME` operation, the debugger (or other host mechanism) has to extract the filename(s) one character at a time from the memory of the target. However, if the bit corresponding to the value `M_STRLEN_PROVIDED` is set, then the I/O control block contains the length of the filename(s) and the debugger is able to use this information to perform a batch read of the target memory (see the description of the fields `fileID` and `nCompleted`).

For a SEEK request, the `flags` field indicates the seek mode (`whence`) as follows:

```
enum
{
   M_SEEK_SET = 0x0001,   /* seek origin is the start of
                             the file                   */
   M_SEEK_CUR = 0x0002,   /* seek origin is the current
                             position within the file   */
```

```
   M_SEEK_END = 0x0004,   /* seek origin is the end of
                             the file               */
};
```

The `flags` field is unused for a CLOSE request.

The `buf` field contains a pointer to the file name for an OPEN or REMOVE request, or a pointer to the data buffer for a READ or WRITE request. For a RENAME operation, this field contains a pointer to the old file name.

The `nDesired` field is set to the number of bytes that should be transferred for a READ or WRITE request. This field is also used by a RENAME request, and is set to a pointer to the new file name.

For a SEEK request, the `nDesired` field contains the offset at which the file should be positioned, relative to the origin specified by the `flags` field. (On architectures that only support 16-bit `ints`, the 32-bit offset at which the file should be positioned is stored in the combined fields `[buf, nDesired]`).

The `nCompleted` field is set by `__primIO` to the number of bytes actually transferred by a `READ` or `WRITE` operation. For a `SEEK` operation, `__primIO` sets this field to the new value of the file pointer. (On architectures that only support 16-bit `ints`, `__primIO` sets the new value of the file pointer in the combined fields `[nCompleted, more]`).

The `RENAME` operation may also make use of the `nCompleted` field. If the operation can determine the lengths of the old and new filenames, then it should store these sizes in the fields `fileID` and `nCompleted`, respectively, and also set the bit-field `flags` to `M_STRLEN_PROVIDED`. The debugger (or other host mechanism) can then use this information to perform a batch read of the target memory to extract the filenames. If this information is not provided, then each character of the file names will have to be read individually.

The `more` field is reserved for future use and currently is always set to `NULL` before calling `_primIO`.

Documented Library Functions

The C run-time library has several categories of functions and macros defined by the ANSI C standard, plus extensions provided by Analog Devices.

The following tables list the library functions documented in this chapter. Note that the tables list the functions for each header file separately; however, the reference pages for these library functions present the functions in alphabetical order.

Table 1-19 lists the library functions in the `ctype.h` header file. Refer to "ctype.h" on page 1-21 for more information on this header file.

Table 1-19. Library Functions in the `ctype.h` Header File

isalnum	isalpha	iscntrl
isdigit	isgraph	islower
isprint	ispunct	isspace
isupper	isxdigit	tolower
toupper		

Table 1-20 lists the library functions in the `locale.h` header file. Refer to "locale.h" on page 1-24 for more information on this header file.

Table 1-20. Library Functions in the `locale.h` Header File

localeconv	setlocale	

Table 1-21 lists the library functions in the `math.h` header file. Refer to "math.h" on page 1-25 for more information on this header file.

Table 1-21. Library Functions in the `math.h` Header File

acos	asin	atan
atan2	ceil	cos
cosh	exp	fabs
floor	fmod	frexp
isinf	isnan	ldexp
log	log10	modf
pow	sin	sinh
sqrt	tan	tanh

Table 1-22 lists the library functions in the `setjmp.h` header file. Refer to "setjmp.h" on page 1-26 for more information on this header file.

Table 1-22. Library Functions in the `setjmp.h` Header File

longjmp	setjmp	

Table 1-23 lists the library functions in the `signal.h` header file. Refer to "signal.h" on page 1-26 for more information on this header file.

Table 1-23. Library Functions in the `signal.h` Header File

clear_interrupt	interrupt	raise
signal		

Table 1-24 lists the library functions in the `stdarg.h` header file. Refer to "stdarg.h" on page 1-27 for more information on this header file.

Table 1-24. Library Functions in the `stdarg.h` Header File

va_arg	va_end	va_start

Table 1-25 lists the library functions in the `stdfix.h` header file. Refer to "stdfix.h" on page 1-27 for more information on this header file.

Table 1-25. Library Functions in the `stdfix.h` Header File

absfx	bitsfx	countlsfx
divifx	fxbits	fxdivi
idivfx	mulifx	roundfx
strtofxfx		

Table 1-26 lists the library functions in the `stdio.h` header file. Refer to "stdio.h" on page 1-30 for more information on this header file.

Table 1-26. Library Functions in the `stdio.h` Header File

clearerr	fclose	feof
ferror	fflush	fgetc
fgetpos	fgets	fopen
fprintf	fputc	fputs
fread	freopen	fscanf
fseek	fsetpos	ftell
fwrite	getc	getchar
gets	perror	printf
putc	putchar	puts
remove	rename	rewind

Table 1-26. Library Functions in the `stdio.h` Header File (Cont'd)

scanf	setbuf	setvbuf
snprintf	sprintf	sscanf
ungetc	vfprintf	vprintf
vsnprintf	vsprintf	

Table 1-27 lists the library functions in the `stdlib.h` header file. Refer to "stdlib.h" on page 1-33 for more information on this header file.

Table 1-27. Library Functions in the `stdlib.h` Header File

abort	abs	atexit
atof	atoi	atol
atold	atoll	avg
bsearch	calloc	clip
count_ones	div	exit
free	getenv	heap_calloc
heap_free	heap_install	heap_lookup_name
heap_malloc	heap_realloc	heap_switch
labs	lavg	lclip
lcount_ones	ldiv	llabs
llavg	llclip	llcount_ones
lldiv	llmax	llmin
lmax	lmin	malloc
max	min	qsort
rand	realloc	set_alloc_type
srand	strtod	strtol
strtold	strtoll	strtoul
strtoull	system	

Table 1-28 lists the library functions in the `string.h` header file. Refer to "string.h" on page 1-35 for more information on this header file.

Table 1-28. Library Functions in the `string.h` Header File

memchr	memcmp	memcpy
memmove	memset	strcat
strchr	strcmp	strcoll
strcpy	strcspn	strerror
strlen	strncat	strncmp
strncpy	strpbrk	strrchr
strspn	strstr	strtok
strxfrm		

Table 1-29 lists the library functions in the `time.h` header file. Refer to "time.h" on page 1-35 for more information on this header file.

Table 1-29. Library Functions in the `time.h` Header File

asctime	clock	ctime
difftime	gmtime	localtime
mktime	strftime	time

C Run-Time Library Reference

The C run-time library is a collection of functions that you can call from your C/C++ programs. This section lists the functions in alphabetical order.

Notation Conventions

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

Reference Format

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

Name and purpose of the function

Synopsis – Required header file and functional prototype

Description – Function specification

Error Conditions – Method that the functions use to indicate an error

Example –Typical function usage

See Also – Related functions

abort

Abnormal program end

Synopsis

```
#include <stdlib.h>
void abort (void);
```

Description

The abort function causes an abnormal program termination by raising the SIGABRT exception. If the SIGABRT handler returns, abort() calls exit() to terminate the program with a failure condition.

Error Conditions

The abort function does not return.

Example

```
#include <stdlib.h>

extern int errors;

if (errors)        /* terminate program if */
    abort();       /* errors are present   */
```

See Also

atexit, exit

abs

Absolute value

Synopsis

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

Description

The `abs` function returns the absolute value of its integer argument.

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, absfx, labs, llabs

absfx

absolute value

Synopsis

```
#include <stdfix.h>

short fract abshr(short fract f);
fract absr(fract f);
long fract abslr(long fract f);
```

Description

The abs*fx* family of functions return the absolute value of their fixed-point input.

In addition to the individually-named functions for each fixed-point type, a type-generic macro `absfx` is defined for use in C99 mode. This may be used with any of the fixed-point types and returns a result of the same type as its operand.

Error Conditions

The `absfx` family of functions do not return an error condition.

Example

```
#include <stdfix.h>
long fract f;
f = abslr(0.75lr);          /* f == 0.75lr */
#if defined(_C99)
f = absfx(0.75lr);          /* f == 0.75lr */
#endif
```

See Also

abs, fabs, labs, llabs

acos

Arc cosine

Synopsis

```
#include <math.h>

float acosf (float x);
double acos (double x);
long double acosd (long double x);
```

Description

The arc cosine functions return the arc cosine of x. The input must be in the range [-1, 1]. The output, in radians, is in the range [0, π].

Error Conditions

The arc cosine functions indicate a domain error (set errno to EDOM) and return a zero if the input is not in the range [-1, 1].

Example

```
#include <math.h>

double x;
float y;

x = acos (0.0);      /* x = π/2 */
y = acosf (0.0);     /* y = π/2 */
```

See Also

cos

asctime

Convert broken-down time into a string

Synopsis

```
#include <time.h>
char *asctime(const struct tm *t);
```

Description

The `asctime` function converts a broken-down time, as generated by the functions `gmtime` and `localtime`, into an ASCII string that will contain the date and time in the form

```
DDD MMM dd hh:mm:ss YYYY\n
```

where

- `DDD` represents the day of the week (that is, Mon, Tue, Wed, and so on)
- `MMM` is the month and will be of the form Jan, Feb, Mar, and so on
- `dd` is the day of the month, from 1 to 31
- `hh` is the number of hours after midnight, from 0 to 23
- `mm` is the minute of the day, from 0 to 59
- `ss` is the second of the day, from 0 to 61 (to allow for leap seconds)
- `YYYY` represents the year

The function returns a pointer to the ASCII string, which may be overwritten by a subsequent call to this function. Also note that the function `ctime` returns a string that is identical to

```
asctime(localtime(&t))
```

Error Conditions

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

Example

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

struct tm tm_date;

printf("The date is %s",asctime(&tm_date));
```

See Also

ctime, gmtime, localtime

asin

Arc sine

Synopsis

```
#include <math.h>

float asinf (float x);
double asin (double x);
long double asind (long double x);
```

Description

The arc sine functions return the arc sine of the first argument. The input must be in the range [-1, 1]. The output, in radians, is in the range $-\pi/2$ to $\pi/2$.

Error Conditions

The arc sine functions indicate a domain error (set `errno` to `EDOM`) and return a zero if the input is not in the range [-1, 1].

Example

```
#include <math.h>

double y;
float x;

y = asin (1.0);       /* y = π/2 */
x = asinf (1.0);      /* x = π/2 */
```

See Also

sin

atan

Arc tangent

Synopsis

```
#include <math.h>

float atanf (float x);
double atan (double x);
long double atand (long double x);
```

Description

The arc tangent functions return the arc tangent of the first argument. The output, in radians, is in the range -π/2 to π/2.

Error Conditions

The arc tangent functions do not return error conditions.

Example

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

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

See Also

atan2, tan

atan2

Arc tangent of quotient

Synopsis

```
#include <math.h>

float atan2f (float y, float x);
double atan2 (double y, double x);
long double atan2d (long double y, long double x);
```

Description

The `atan2` functions compute the arc tangent of the input value `y` divided by input value `x`. The output, in radians, is in the range -π to π.

Error Conditions

The `atan2` functions return a zero if `x=0` and `y=0`.

Example

```
#include <math.h>

double a,d;
float b,c;

a = atan2 (0.0, 0.0);       /* the error condition: a = 0.0 */
b = atan2f (1.0, 1.0);      /* b = π/4                      */

c = atan2f (1.0, 0.0);      /* c = π/2  */
d = atan2 (-1.0, 0.0);      /* d = -π/2 */
```

See Also

atan, tan

atexit

Register a function to call at program termination

Synopsis

```
#include <stdlib.h>
int atexit (void (*func)(void));
```

Description

The `atexit` function registers a function to be called at program termination. Functions are called once for each time they are registered, in the reverse order of registration. Up to 32 functions can be registered using the `atexit` function.

Error Conditions

The `atexit` function returns a non-zero value if the function cannot be registered.

Example

```
#include <stdlib.h>

extern void goodbye(void);

if (atexit(goodbye))
   exit(1);
```

See Also

abort, exit

atof

Convert string to a double

Synopsis

```
#include <stdlib.h>
double atof(const char *nptr);
```

Description

The `atof` function converts a character string into a floating-point value of type `double`, and returns its value. The character string is pointed to by the argument `nptr` and may contain any number of leading whitespace characters (as determined by the function `isspace`) followed by a floating-point number. The floating-point number may either be a decimal floating-point number or a hexadecimal floating-point number.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (+) or minus (-); and `digits` are one or more decimal digits. The sequence of digits may contain a decimal point (.).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (-) followed by the hexadecimal prefix `0x` or `0X`. This character

sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P`, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number stops the scan.

Error Conditions

The `atof` function returns a zero if no conversion could be made. If the correct value results in an overflow, a positive or negative (as appropriate) `HUGE_VAL` is returned. If the correct value results in an underflow, 0.0 is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

Notes

The `atof (pdata)` function reference is functionally equivalent to:

```
strtod (pdata, (char *) NULL);
```

and therefore, if the function returns zero, it is not possible to determine whether the character string contained a (valid) representation of 0.0 or some invalid numerical string.

Example

```
#include <stdlib.h>

double x;

x = atof("5.5");       /* x = 5.5 */
```

See Also

atoi, atol, atoll, strtod

atoi

Convert string to integer

Synopsis

```
#include <stdlib.h>
int atoi (const char *nptr);
```

Description

The `atoi` function converts a character string to an integer value. The character string to be converted is pointed to by the input pointer, `nptr`. The function clears any leading characters for which `isspace` would return true. Conversion begins at the first digit (with an optional preceding sign) and terminates at the first non-digit.

Error Conditions

The `atoi` function returns -1 if no conversion can be made.

Example

```
#include <stdlib.h>

int i;

i = atoi ("5");      /* i = 5 */
```

See Also

atof, atol, atoll, strtod

atol

Convert string to long integer

Synopsis

```
#include <stdlib.h>
long atol (const char *nptr);
```

Description

The `atol` function converts a character string to a long integer value. The character string to be converted is pointed to by the input pointer, `nptr`. The function clears any leading characters for which `isspace` would return true. Conversion begins at the first digit (with an optional preceding sign) and terminates at the first non-digit.

There is no way to determine if a zero is a valid result or an indicator of an invalid string.

Error Conditions

The `atol` function returns -1 if no conversion can be made.

Example

```
#include <stdlib.h>

long int i;

i = atol ("5");      /* i = 5 */
```

See Also

atof, atoi, atoll, strtod, strtol, strtoll, strtoul, strtoull

atold

Convert string to a long double

Synopsis

```
#include <stdlib.h>
long double atold(const char *nptr);
```

Description

The `atold` function is an extension to the ISO/IEC 9899:1990 C standard and the ISO/IEC 9899:1999 C standard.

The `atold` function converts a character string into a floating-point value of type `long double`, and returns its value. The character string is pointed to by the argument `nptr` and may contain any number of leading whitespace characters (as determined by the function `isspace`) followed by a floating-point number. The floating-point number may either be a decimal floating-point number or a hexadecimal floating-point number.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (+) or minus (-); and `digits` are one or more decimal digits. The sequence of digits may contain a decimal point (.).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (-) followed by the hexadecimal prefix `0x` or `0X` . This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P` , an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number stops the scan.

Error Conditions

The `atold` function returns a zero if no conversion could be made. If the correct value results in an overflow, a positive or negative (as appropriate) `LDBL_MAX` is returned. If the correct value results in an underflow, 0.0 is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

Notes

The `atold (pdata)` function reference is functionally equivalent to:

```
strtold (pdata, (char *) NULL);
```

and therefore, if the function returns zero, it is not possible to determine whether the character string contained a (valid) representation of 0.0 or some invalid numerical string.

Example

```
#include <stdlib.h>

long double x;

x = atold("5.5");      /* x = 5.5 */
```

See Also

atol, atoi, atoll, strtold

atoll

Convert string to long long integer

Synopsis

```
#include <stdlib.h>
long long atoll (const char *nptr);
```

Description

The `atoll` function converts a character string to a `long long` integer value. The character string to be converted is pointed to by the input pointer, `nptr`. The function clears any leading characters for which `isspace` would return true. Conversion begins at the first digit (with an optional preceding sign) and terminates at the first non-digit.

ⓘ There is no way to determine if a zero is a valid result or an indicator of an invalid string.

Error Conditions

The `atoll` function returns 0 if no conversion can be made.

Example

```
#include <stdlib.h>

long long i;

i = atoll ("150000000000000");  /* i = 150000000000000LL */
```

See Also

atof, atoi, atol, strtod, strtol, strtoll, strtoul, strtoull

avg

Mean of two values

Synopsis

```
#include <stdlib.h>
int avg (int x, int y);
```

Description

The avg function is an Analog Devices extension to the ANSI standard.

The avg function adds two arguments and divides the result by two. The avg function is a built-in function which is implemented with an Rn=(Rx+Ry)/2 instruction.

Error Conditions

The avg function does not return an error code.

Example

```
#include <stdlib.h>

int i;

i = avg (10, 8);      /* returns 9 */
```

See Also

lavg, llavg

bitsfx

Bitwise fixed-point to integer conversion

Synopsis

```
#include <stdfix.h>

int_hr_t bitshr(short fract f);
int_r_t bitsr(fract f);
int_lr_t bitslr(long fract f);
uint_uhr_t bitsuhr(unsigned short fract f);
uint_ur_t bitsur(unsigned fract f);
uint_ulr_t bitsulr(unsigned long fract f);
```

Description

Given a fixed-point operand, the `bits`*fx* family of functions return the fixed-point value multiplied by 2^F, where `F` is the number of fractional bits in the fixed-point type. This is equivalent to the bit-pattern of the fixed-point value held in an integer type.

Error Conditions

The `bits`*fx* family of functions do not return an error condition.

Example

```
#include <stdfix.h>
uint_ulr_t ulr;
ulr = bitsulr(0.125ulr);       /* ulr == 0x20000000 */
```

See Also

fxbits

bsearch

Perform binary search in a sorted array

Synopsis

```
#include <stdlib.h>

void *bsearch (const void *key, const void *base,
               size_t nelem, size_t size,
               int (*compare)(const void *, const void *));
```

Description

The `bsearch` function searches the array `base` for an array element that matches the element `key`. The size of each array element is specified by `size`, and the array is defined to have `nelem` array elements.

The `bsearch` function will call the function `compare` with two arguments; the first argument will point to the array element `key` and the second argument will point to an element of the array. The `compare` function should return an integer that is either zero, or less than zero, or greater than zero, depending upon whether the array element `key` is equal to, less than, or greater than the array element pointed to by the second argument.

If the comparison function returns a zero, then `bsearch` will return a pointer to the matching array element; if there is more than one matching elements then it is not defined which element is returned. If no match is found in the array, `bsearch` will return `NULL`.

The array to be searched would normally be sorted according to the criteria used by the comparison function (the `qsort` function may be used to first sort the array if necessary).

Error Conditions

The `bsearch` function returns a null pointer when the key is not found in the array.

Example

```
#include <stdlib.h>
#include <string.h>
#define SIZE 3

struct record_t {
    char *name;
    char *street;
    char *city;
};

struct record_t data_base[SIZE] = {
    {"Baby Doe" , "Central Park" , "New York"},
    {"Jane Doe" , "Regents Park" , "London"  },
    {"John Doe" , "Queens Park"  , "Sydney"  }
};

static int
compare_function (const void *arg1, const void *arg2)
{
    const struct record_t *pkey  = arg1;
    const struct record_t *pbase = arg2;

    return strcmp (pkey->name,pbase->name);
}
```

```
struct record_t key = {"Baby Doe" , "" , ""};
struct record_t *search_result;

search_result = bsearch (&key,
                         data_base,
                         SIZE,
                         sizeof(struct record_t),
                         compare_function);
```

See Also

qsort

calloc

Allocate and initialize memory

Synopsis

```
#include <stdlib.h>
void *calloc (size_t nmemb, size_t size);
```

Description

The `calloc` function dynamically allocates a range of memory and initializes all locations to zero. The number of elements (the first argument) multiplied by the size of each element (the second argument) is the total memory allocated. The memory may be deallocated with the `free` function.

The object is allocated from the current heap, which is the default heap unless `set_alloc_type` or `heap_switch` has been called to change the current heap to an alternate heap.

Error Conditions

The `calloc` function returns a null pointer if unable to allocate the requested memory.

Example

```
#include <stdlib.h>

int *ptr;

ptr = (int *) calloc (10, sizeof (int));
      /* ptr points to a zeroed array of length 10 */
```

See Also

free, heap_calloc, heap_free, heap_lookup_name, heap_malloc, heap_realloc, malloc, realloc, set_alloc_type

ceil

Ceiling

Synopsis

```
#include <math.h>

float ceilf (float x);
double ceil (double x);
long double ceild (long double x);
```

Description

The ceiling functions return the smallest integral value that is not less than the argument `x`.

Error Conditions

The ceiling functions do not return an error condition.

Example

```
#include <math.h>

double y;
float x;

y = ceil (1.05);       /* y = 2.0  */
x = ceilf (-1.05);     /* y = -1.0 */
```

See Also

floor

clear_interrupt

Clear a pending signal

Synopsis

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

Description

The `clear_interrupt` function is an Analog Devices extension to the ANSI standard.

The `clear_interrupt` function clears the signal `sig` in the `IRPTL` register. Table 1-30, Table 1-31 on page 1-109, Table 1-32 on page 1-110, Table 1-33 on page 1-112, Table 1-35 on page 1-115, and Table 1-35 on page 1-115 show the possible values that the `sig` argument may be set to for the appropriate ADSP-21xxx processor.

The `clear_interrupt` function does not work for interrupts that set any status bits in the `STKY` register, such as floating-point overflow.

Table 1-30. ADSP-21020 Processor Signals

SIG Value	Description
SIG_SOVF	Status stack or Loop stack overflow or PC stack full
SIG_TMZ0	Timer = 0 (high priority option)
SIG_IRQ3	Interrupt 3
SIG_IRQ2	Interrupt 2
SIG_IRQ1	Interrupt 1
SIG_IRQ0	Interrupt 0
SIG_CB7	Circular buffer 7 overflow
SIG_CB15	Circular buffer 15 overflow
SIG_TMZ	Timer = 0 (low priority option)

Table 1-30. ADSP-21020 Processor Signals (Cont'd)

SIG Value	Description
SIG_FIX	Fixed-point overflow
SIG_FLTO	Floating-point overflow exception
SIG_FLTU	Floating-point underflow exception
SIG_FLTI	Floating-point invalid exception
SIG_USR0	User software interrupt 0
SIG_USR1	User software interrupt 1
SIG_USR2	User software interrupt 2
SIG_USR3	User software interrupt 3
SIG_USR4	User software interrupt 4
SIG_USR5	User software interrupt 5

Table 1-31. ADSP-2106x Processor Signals

SIG Value	Definition
SIG_SOVF	Status stack or Loop stack overflow or PC stack full
SIG_TMZ0	Timer = 0 (high priority option)
SIG_VIRPTI	Vector Interrupt
SIG_IRQ2	Interrupt 2
SIG_IRQ1	Interrupt 1
SIG_IRQ0	Interrupt 0
SIG_SPR0I	DMA Channel 0 - SPORT0 Receive
SIG_SPR1I	DMA Channel 1 - SPORT1 Receive (or Link Buffer 0)
SIG_SPT0I	DMA Channel 2 - SPORT0 Transmit
SIG_SPT1I	DMA Channel 3 - SPORT1 Transmit (or Link Buffer 1)
[1]SIG_LP2I	DMA Channel 4 - Link Buffer 2
[1]SIG_LP3I	DMA Channel 5 - Link Buffer 3
SIG_EP0I	DMA Channel 6 - Ext. Port Buffer 0 (or Link Buffer 4)

Table 1-31. ADSP-2106x Processor Signals (Cont'd)

SIG Value	Definition
SIG_EP1I	DMA Channel 7 - Ext. Port Buffer 1 (or Link Buffer 5)
[1]SIG_EP2I	DMA Channel 8 - Ext. Port Buffer 2
[1]SIG_EP3I	DMA Channel 9 - Ext. Port Buffer 3
[1]SIG_LSRQ	Link port service request
SIG_CB7	Circular buffer 7 overflow
SIG_CB15	Circular buffer 15 overflow
SIG_TMZ	Timer = 0 (low priority option)
SIG_FIX	Fixed-point overflow
SIG_FLTO	Floating-point overflow exception
SIG_FLTU	Floating-point underflow exception
SIG_FLTI	Floating-point invalid exception
SIG_USR0	User software interrupt 0
SIG_USR1	User software interrupt 1
SIG_USR2	User software interrupt 2
SIG_USR3	User software interrupt 3

1 Signal is not present on the ADSP-21061 and ADSP-21065L processors.

Table 1-32. ADSP-2116x Processor Signals

SIG Value	Definition	Processor Restrictions
SIG_IICDI	Illegal input condition detected	
SIG_SOVF	Status stack or Loop stack overflow or PC stack full	
SIG_TMZ0	Timer = 0 (high priority option)	
SIG_VIRPTI	Vector interrupt	
SIG_IRQ2	Interrupt 2	
SIG_IRQ1	Interrupt 1	

Table 1-32. ADSP-2116x Processor Signals (Cont'd)

SIG Value	Definition	Processor Restrictions
SIG_IRQ0	Interrupt 0	
SIG_SPR0I	SPORT0 Receive	ADSP-21160 only
SIG_SPR1I	SPORT1 Receive	ADSP-21160 only
SIG_SPT01	SPORT0 Transmit	ADSP-21160 only
SIG_SPT1I	SPORT0 Transmit	ADSP-21160 only
SIG_SP0I	SPORT0 DMA	ADSP-21161 only
SIG_SP1I	SPORT1 DMA	ADSP-21161 only
SIG_SP2I	SPORT2 DMA	ADSP-21161 only
SIG_SP3I	SPORT3 DMA	ADSP-21161 only
SIG_LP0I	Link Buffer 0	
SIG_LP1I	Link Buffer 1	
SIG_LP2I	Link Buffer 2	ADSP-21160 only
SIG_LP3I	Link Buffer 3	ADSP-21160 only
SIG_LP4I	Link Buffer 4	ADSP-21160 only
SIG_LP5I	Link Buffer 5	ADSP-21160 only
SIG_SPIRI	SPI Receive DMA	ADSP-21161 only
SIG_SPITI	SPI Transmit DMA	ADSP-21161 only
SIG_EP0I	Ext. Port Buffer 0	
SIG_EP1I	Ext. Port Buffer 1	
SIG_EP2I	Ext. Port Buffer 2	

Table 1-32. ADSP-2116x Processor Signals (Cont'd)

SIG Value	Definition	Processor Restrictions
SIG_EP3I	Ext. Port Buffer 3	
SIG_LSRQ	Link port service request	
SIG_CB7	Circular buffer 7 overflow	
SIG_CB15	Circular buffer 15 overflow	
SIG_TMZ	Timer = 0 (low priority option)	
SIG_FIX	Fixed-point overflow	
SIG_FLTO	Floating-point overflow exception	
SIG_FLTU	Floating-point underflow exception	
SIG_FLTI	Floating-point invalid exception	
SIG_USR0	User software interrupt 0	
SIG_USR1	User software interrupt 1	

Table 1-33. ADSP-2126x Processor Signals

SIG Value	Definition
SIG_IICDI	Illegal input condition detected
SIG_SOVF	Status stack or Loop stack overflow or PC stack full
SIG_TMZ0	Timer = 0 (high priority option)
SIG_BKP	Hardware breakpoint
SIG_IRQ2	Interrupt 2
SIG_IRQ1	Interrupt 1
SIG_IRQ0	Interrupt 0
SIG_DAIH	DAI High priority
SIG_SPIH	SPI transmit or receive (high priority option)
SIG_GPTMR0	General-purpose IOP timer 0
SIG_SP1	SPORT 1

Table 1-33. ADSP-2126x Processor Signals (Cont'd)

SIG Value	Definition
SIG_SP3	SPORT 3
SIG_SP5	SPORT 5 (ADSP-21262 and ADSP-21266 processors only)
SIG_SP0	SPORT 0
SIG_SP2	SPORT 2
SIG_SP4	SPORT 4 (ADSP-21262 and ADSP-21266 processors only)
SIG_PP	Parallel port
SIG_GPTMR1	General-purpose IOP timer 1
SIG_DAIL	DAI low priority
SIG_GPTMR2	General-purpose IOP timer 2
SIG_SPIL	SPI transmit or receive (low priority option)
SIG_CB7	Circular buffer 7 overflow
SIG_CB15	Circular buffer 15 overflow
SIG_TMZ	Timer = 0 (low priority option)
SIG_FIX	Fixed-point overflow
SIG_FLTO	Floating-point overflow exception
SIG_FLTU	Floating-point underflow exception
SIG_FLTI	Floating-point invalid exception
SIG_USR0	User software interrupt 0
SIG_USR1	User software interrupt 1
SIG_USR2	User software interrupt 2
SIG_USR3	User software interrupt 3

Table 1-34. ADSP-2136x Processor Signals

SIG Value	Definition	Default setting (for programmable peripheral interrupts)
SIG_IICDI	Illegal input condition detected	
SIG_SOVF	Status stack or Loop stack overflow or PC stack full	
SIG_TMZ0	Timer = 0 (high priority option)	
SIG_BKP	Hardware breakpoint	
SIG_IRQ2	Interrupt 2	
SIG_IRQ1	Interrupt 1	
SIG_IRQ0	Interrupt 0	
SIG_P0	Peripheral interrupt - 0	DAI High priority
SIG_P1	Peripheral interrupt - 1	SPI transmit or receive (high priority option)
SIG_P2	Peripheral interrupt - 2	General-purpose IOP timer 0
SIG_P3	Peripheral interrupt - 3	SPORT 1
SIG_P4	Peripheral interrupt - 4	SPORT 3
SIG_P5	Peripheral interrupt - 5	SPORT 5
SIG_P6	Peripheral interrupt - 6	SPORT 0
SIG_P7	Peripheral interrupt - 7	SPORT 2
SIG_P8	Peripheral interrupt - 8	SPORT 4
SIG_P9	Peripheral interrupt - 9	Parallel port
SIG_P10	Peripheral interrupt - 10	General-purpose IOP timer 1
SIG_P12	Peripheral interrupt - 12	DAI low priority
SIG_P13	Peripheral interrupt - 13	PWM
SIG_P15	Peripheral interrupt - 15	DTCP
SIG_P17	Peripheral interrupt - 17	General-purpose IOP timer 2

Table 1-34. ADSP-2136x Processor Signals (Cont'd)

SIG Value	Definition	Default setting (for programmable peripheral interrupts)
SIG_P18	Peripheral interrupt - 18	SPI transmit or receive (low priority option)
SIG_CB7	Circular buffer 7 overflow	
SIG_CB15	Circular buffer 15 overflow	
SIG_TMZ	Timer = 0 (low priority option)	
SIG_FIX	Fixed-point overflow	
SIG_FLTO	Floating-point overflow exception	
SIG_FLTU	Floating-point underflow exception	
SIG_FLTI	Floating-point invalid exception	
SIG_USR0	User software interrupt 0	
SIG_USR1	User software interrupt 1	
SIG_USR2	User software interrupt 2	
SIG_USR3	User software interrupt 3	

Table 1-35. ADSP-214xx Processor Signals

SIG Value	Definition	Default setting (for programmable peripheral interrupts)
SIG_IICDI	Illegal input condition detected	
SIG_SOVF	Status stack or Loop stack overflow or PC stack full	
SIG_TMZ0	Timer = 0 (high priority option)	
SIG_BKP	Hardware breakpoint	
SIG_IRQ2	Interrupt 2	
SIG_IRQ1	Interrupt 1	
SIG_IRQ0	Interrupt 0	

Table 1-35. ADSP-214xx Processor Signals (Cont'd)

SIG Value	Definition	Default setting (for programmable peripheral interrupts)
SIG_P0	Peripheral interrupt - 0	DAI High priority
SIG_P1	Peripheral interrupt - 1	SPI transmit or receive (high priority option)
SIG_P2	Peripheral interrupt - 2	General-purpose IOP timer 0
SIG_P3	Peripheral interrupt - 3	SPORT 1
SIG_P4	Peripheral interrupt - 4	SPORT 3
SIG_P5	Peripheral interrupt - 5	SPORT 5
SIG_P6	Peripheral interrupt - 6	SPORT 0
SIG_P7	Peripheral interrupt - 7	SPORT 2
SIG_P8	Peripheral interrupt - 8	SPORT 4
SIG_P9	Peripheral interrupt - 9	Parallel port
SIG_P10	Peripheral interrupt - 10	General-purpose IOP timer 1
SIG_P12	Peripheral interrupt - 12	DAI low priority
SIG_P13	Peripheral interrupt - 13	PWM
SIG_P15	Peripheral interrupt - 15	DTCP
SIG_P17	Peripheral interrupt - 17	General-purpose IOP timer 2
SIG_P18	Peripheral interrupt - 18	SPI transmit or receive (low priority option)
SIG_CB7	Circular buffer 7 overflow	
SIG_CB15	Circular buffer 15 overflow	
SIG_TMZ	Timer = 0 (low priority option)	
SIG_FIX	Fixed-point overflow	
SIG_FLTO	Floating-point overflow exception	
SIG_FLTU	Floating-point underflow exception	

Table 1-35. ADSP-214xx Processor Signals (Cont'd)

SIG Value	Definition	Default setting (for programmable peripheral interrupts)
SIG_FLTI	Floating-point invalid exception	
SIG_USR0	User software interrupt 0	
SIG_USR1	User software interrupt 1	
SIG_USR2	User software interrupt 2	
SIG_USR3	User software interrupt 3	

Error Conditions

The `clear_interrupt` function returns a 1 if the interrupt was pending; otherwise 0 is returned.

Example

```
#include <signal.h>
clear_interrupt (SIG_IRQ2);
   /* clear the interrupt 2 latch */
```

See Also

interrupt, raise, signal

clearerr

Clear file or stream error indicator

Synopsis

```
#include <stdio.h>
void clearerr(FILE *stream);
```

Description

The `clearerr` function clears the error and end-of-file (`EOF`) indicators for the particular stream pointed to by `stream`.

The `stream` error indicators record whether any read or write errors have occurred on the associated stream. The `EOF` indicator records when there is no more data in the file.

Error Conditions

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

Example

```
#include <stdio.h>

FILE *routine(char *filename)
{
   FILE *fp;
   fp = fopen(filename, "r");
   /* Some operations using the file */
   /* now clear the error indicators for the stream */
   clearerr(fp);
   return fp;
}
```

See Also

feof, ferror

clip

Clip

Synopsis

```
#include <stdlib.h>
int clip (int value1, int value2);
```

Description

The `clip` function is an Analog Devices extension to the ANSI standard.

The `clip` function returns its first argument if its absolute value is less than the absolute value of its second argument, otherwise it returns the absolute value of its second argument if the first is positive, or minus the absolute value if the first argument is negative. The `clip` function is a built-in function which is implemented with an `Rn = CLIP Rx BY Ry` instruction.

Error Conditions

The `clip` function does not return an error code.

Example

```
#include <stdlib.h>

int i;

i = clip (10, 8);       /* returns 8  */
i = clip (8, 10);       /* returns 8  */
i = clip (-10, 8);      /* returns -8 */
```

See Also

lclip, llclip

clock

Processor time

Synopsis

```
#include <time.h>
clock_t clock(void);
```

Description

The `clock` function returns the number of processor cycles that have elapsed since an arbitrary starting point. The function returns the value `(clock_t) -1`, if the processor time is not available or if it cannot be represented. The result returned by the function may be used to calculate the processor time in seconds by dividing it by the macro `CLOCKS_PER_SEC`. For more information, see "time.h" on page 1-35. An alternative method of measuring the performance of an application is described in "Measuring Cycle Counts" on page 1-48.

Error Conditions

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

Example

```
#include <time.h>

time_t start_time,stop_time;
double time_used;

start_time = clock();
compute();
stop_time = clock();

time_used = ((double) (stop_time - start_time)) / CLOCKS_PER_SEC;
```

See Also

No related function.

cos

Cosine

Synopsis

```
#include <math.h>

float cosf (float x);
double cos (double x);
long double cosd (long double x);
```

Description

The cosine functions return the cosine of the first argument. The input is interpreted as radians; the output is in the range [-1, 1].

Error Conditions

The input argument `x` for `cosf` must be in the domain [-1.647e6, 1.647e6] and the input argument for `cosd` must be in the domain [-8.433e8, 8.433e8]. The functions return zero if `x` is outside their domain.

Example

```
#include <math.h>

double y;
float x;

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

See Also

acos, sin

cosh

Hyperbolic cosine

Synopsis

```
#include <math.h>

float coshf (float x);
double cosh (double x);
long double coshd (long double x);
```

Description

The hyperbolic cosine functions return the hyperbolic cosine of their argument.

Error Conditions

The domain of `coshf` is [-89.39, 89.39], and the domain for `coshd` is [-710.44, 710.44]. The functions return `HUGE_VAL` if the input argument `x` is outside the respective domains.

Example

```
#include <math.h>

float  x;
double y;

x = coshf ( 1.0);    /* x = 1.54308 */
y = cosh  (-1.0);    /* y = 1.54308 */
```

See Also

sinh

count_ones

Count one bits in word

Synopsis

```
#include <stdlib.h>
int count_ones (int value);
```

Description

The `count_ones` function is an Analog Devices extension to the ANSI standard.

The `count_ones` function returns the number of one bits in its argument.

Error Conditions

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

Example

```
#include <stdlib.h>

int flags1 = 0xAD1;
int flags2 = -1;
int cnt1;
int cnt2;

cnt1 = count_ones (flags1);    /* returns 6 */
cnt2 = count_ones (flags2);    /* returns 32 */
```

See Also

lcount_ones, llcount_ones

countlsfx

Count leading sign or zero bits

Synopsis

```
#include <stdfix.h>

int countlshr(short fract f);
int countlsr(fract f);
int countlslr(long fract f);

int countlsuhr(unsigned short fract f);
int countlsur(unsigned fract f);
int countlsulr(unsigned long fract f);
```

Description

Given a fixed-point operand `x`, the `countls`*`fx`* family of functions return the largest value of `n` for which `x << n` does not overflow. For a zero input value, the function will return the number of bits in the fixed-point type.

In addition to the individually-named functions for each fixed-point type, a type-generic macro `countlsfx` is defined for use in C99 mode. This may be used with any of the fixed-point types.

Error Conditions

The `countls`*`fx`* family of functions do not return an error condition.

Example

```
#include <stdfix.h>
int n;
n = countlsulr(0.125ulr);        /* n == 2 */
#if defined(_C99)
n = countlsfx(0.125ulr);         /* n == 2 */
#endif
```

See Also

No related functions.

ctime

Convert calendar time into a string

Synopsis

```
#include <time.h>
char *ctime(const time_t *t);
```

Description

The `ctime` function converts a calendar time, pointed to by the argument `t` into a string that represents the local date and time. The form of the string is the same as that generated by `asctime`, and so a call to `ctime` is equivalent to

```
asctime(localtime(&t))
```

A pointer to the string is returned by `ctime`, and it may be overwritten by a subsequent call to the function.

Error Conditions

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

Example

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

time_t cal_time;

if (cal_time != (time_t)-1)
    printf("Date and Time is %s",ctime(&cal_time));
```

See Also

asctime, gmtime, localtime, time

difftime

Difference between two calendar times

Synopsis

```
#include <time.h>
double difftime(time_t t1, time_t t0);
```

Description

The `difftime` function returns the difference in seconds between two calendar times, expressed as a `double`. By default, the `double` data type represents a 32-bit, single precision, floating-point, value. This form is normally insufficient to preserve all of the bits associated with the difference between two calendar times, particularly if the difference represents more than 97 days. It is recommended therefore that any function that calls `difftime` is compiled with the `-double-size-64` switch.

Error Conditions

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

Example

```
#include <time.h>
#include <stdio.h>
#define NA ((time_t)(-1))

time_t cal_time1;
time_t cal_time2;
double time_diff;
```

```
if ((cal_time1 == NA) || (cal_time2 == NA))
    printf("calendar time difference is not available\n");
else
    time_diff = difftime(cal_time2,cal_time1);
```

See Also

time

div

Division

Synopsis

```
#include <stdlib.h>
div_t div (int numer, int denom);
```

Description

The `div` function divides `numer` by `denom`, both of type `int`, and returns a structure of type `div_t`. The type `div_t` is defined as:

```
typedef struct {
   int quot;
   int rem;
} div_t;
```

where `quot` is the quotient of the division and `rem` is the remainder, such that if `result` is of type `div_t`, then

```
result.quot * denom + result.rem == numer
```

Error Conditions

If `denom` is zero, the behavior of the `div` function is undefined.

Example

```
#include <stdlib.h>

div_t result;

result = div (5, 2);    /* result.quot = 2, result.rem = 1 */
```

See Also

divifx, fmod, fxdivi, idivfx, ldiv, lldiv, modf

divifx

Division of integer by fixed-point to give integer result

Synopsis

```
#include <stdfix.h>

int divir(int numer, fract denom);
long int divilr(long int numer, long fract denom);
unsigned int diviur(unsigned int numer, unsigned fract denom);
unsigned long int diviulr(unsigned long int numer,
                          unsigned long fract denom);
```

Description

Given an integer numerator and a fixed-point denominator, the `divi`*`fx`* family of functions computes the quotient and returns the closest integer value to the result.

Error Conditions

The `divi`*`fx`* family of functions have undefined behavior if the denominator is zero.

Example

```
#include <stdfix.h>
unsigned long int ulquo;
ulquo = diviulr(125, 0.125ulr);        /* ulquo == 1000 */
```

See Also

div, fxdivi, idivfx, ldiv, lldiv

exit

Normal program termination

Synopsis

```
#include <stdlib.h>
void exit (int status);
```

Description

The `exit` function causes normal program termination. The functions registered by the `atexit` function are called in reverse order of their registration and the processor is put into the `IDLE` state. The `status` argument is stored in register `R0`, and control is passed to the label `___lib_prog_term`, which is defined in the run-time startup file.

Error Conditions

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

Example

```
#include <stdlib.h>

exit (EXIT_SUCCESS);
```

See Also

abort, atexit

exp

Exponential

Synopsis

```
#include <math.h>

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

Description

The exponential functions compute the exponential value `e` to the power of their argument.

Error Conditions

The input argument `x` for `expf` must be in the domain [-87.33, 88.72] and the input argument for `expd` must be in the domain [-708.2, 709.1]. The functions return `HUGE_VAL` if `x` is greater than the domain and 0.0 if `x` is less than the domain.

Example

```
#include <math.h>

double y;
float x;

y = exp (1.0);      /* y = 2.71828 */
x = expf (1.0);     /* x = 2.71828 */
```

See Also

log, pow

fabs

Absolute value

Synopsis

```
#include <math.h>

float fabsf (float x);
double fabs (double x);
long double fabsd (long double x);
```

Description

The `fabs` functions return the absolute value of the argument `x`.

Error Conditions

The `fabs` functions do not return error conditions.

Example

```
#include <math.h>

double y;
float x;

y = fabs (-2.3);      /* y = 2.3 */
y = fabs (2.3);       /* y = 2.3 */
x = fabsf (-5.1);     /* x = 5.1 */
```

See Also

abs, absfx, labs, llabs

fclose

Close a stream

Synopsis

```
#include <stdio.h>
int fclose(FILE *stream);
```

Description

The `fclose` function flushes `stream` and closes the associated file. The flush will result in any unwritten buffered data for the stream to be written to the file, with any unread buffered data being discarded.

If the buffer associated with `stream` was allocated automatically it will be deallocated.

The `fclose` function will return 0 on successful completion.

Error Conditions

If the `fclose` function is not successful it returns `EOF`.

Example

```
#include <stdio.h>

void example(char* fname)
{
   FILE *fp;
   fp = fopen(fname, "w+");
   /* Do some operations on the file */
   fclose(fp);
}
```

See Also

fopen

feof

Test for end of file

Synopsis

```
#include <stdio.h>
int feof(FILE *stream);
```

Description

The `feof` function tests whether or not the file identified by `stream` has reached the end of the file. The routine returns 0 if the end of the file has not been reached and a non-zero result of the end of file has been reached.

Error Conditions

The `feof` function does not return any error condition.

Example

```
#include <stdio.h>

void print_char_from_file(FILE *fp)
{
   /* printf out each character from a file until EOF */
   while (!feof(fp))
      printf("%c", fgetc(fp));
   printf("\n");
}
```

See Also

clearerr, ferror

ferror

Test for read or write errors

Synopsis

```
#include <stdio.h>
int ferror(FILE *stream);
```

Description

The `ferror` function tests whether an uncleared error has occurred while accessing `stream`. If there are no errors then the function will return zero, otherwise it will return a non-zero value.

ⓘ The `ferror` function does not examine whether the file identified by `stream` has reached the end of the file.

Error Conditions

The `ferror` function does not return any error condition.

Example

```
#include <stdio.h>

void test_for_error(FILE *fp)
{
   if (ferror(fp))
      printf("Error with read/write to stream\n");
   else
      printf("read/write to stream OKAY\n");
}
```

See Also

clearerr, feof

fflush

Flush a stream

Synopsis

```
#include <stdio.h>
int fflush(FILE *stream);
```

Description

The `fflush` function causes any unwritten data for `stream` to be written to the file. If `stream` is a `NULL` pointer, `fflush` performs this flushing action on all streams.

Upon successful completion the `fflush` function returns zero.

Error Conditions

If `fflush` is unsuccessful, the `EOF` value is returned.

Example

```
#include <stdio.h>

void flush_all_streams(void)
{
   fflush(NULL);
}
```

See Also

fclose

fgetc

Get a character from a stream

Synopsis

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Description

The `fgetc` function obtains the next character from the input stream pointed to by `stream`, converts it from an `unsigned char` to an `int` and advances the file position indicator for the stream.

If there are no errors, then `fgetc` will return the next character as the function result.

Error Conditions

If the `fgetc` function is unsuccessful, `EOF` is returned.

Example

```
#include <stdio.h>

char use_fgetc(FILE *fp)
{
   char ch;
   if ((ch = fgetc(fp)) == EOF) {
      printf("Read End-of-file\n")
      return 0;
   } else {
      return ch;
   }
}
```

See Also

getc

fgetpos

Record the current position in a stream

Synopsis

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Description

The `fgetpos` function stores the current value of the file position indicator for the stream pointed to by `stream` in the file position type object pointed to by `pos`. The information generated by `fgetpos` in `pos` can be used with the `fsetpos` function to return the file to this position.

Upon successful completion the `fgetpos` function will return 0.

Error Conditions

If `fgetpos` is unsuccessful, the function will return a non-zero value.

Example

```
#include <stdio.h>

void aroutine(FILE *fp, char *buffer)
{
   fpos_t pos;
   /* get the current file position */
   if (fgetpos(fp, &pos)!= 0) {
      printf("fgetpos failed\n");
      return;
   }
   /* write the buffer to the file */
   (void) fprintf(fp, "%s\n", buffer);
   /* reset the file position to the value before the write */
```

```
    if (fsetpos(fp, &pos) != 0) {
       printf("fsetpos failed\n");
    }
 }
```

See Also

fsetpos, ftell, fseek, rewind

fgets

Get a string from a stream

Synopsis

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

Description

The `fgets` function reads characters from `stream` into the array pointed to by `s`. The function will read a maximum of one less character than the value specified by `n`, although the get will also end if either a `NEWLINE` character or the end-of-file marker are read. The array `s` will have a `NUL` character written at the end of the string that has been read.

Upon successful completion the `fgets` function will return `s`.

Error Conditions

If `fgets` is unsuccessful, the function will return a `NULL` pointer.

Example

```
#include <stdio.h>

char buffer[20];
void read_into_buffer(FILE *fp)
{
   char *str;

   str = fgets(buffer, sizeof(buffer), fp);
```

```
    if (str == NULL) {
        printf("Either read failed or EOF encountered\n");
    } else {
        printf("filled buffer with %s\n", str);
    }
}
```

See Also

fgetc, getc, gets

floor

Floor

Synopsis

```
#include <math.h>

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

Description

The `floor` functions return the largest integral value that is not greater than their argument.

Error Conditions

The `floor` functions do not return error conditions.

Example

```
#include <math.h>

double y;
float z;

y = floor (1.25);      /* y = 1.0  */
y = floor (-1.25);     /* y = -2.0 */
z = floorf (10.1);     /* z = 10.0 */
```

See Also

ceil

fmod

Floating-point modulus

Synopsis

```
#include <math.h>

float fmodf (float x, float y);
double fmod (double x, double y);
long double fmodd (long double x, long double y);
```

Description

The `fmod` functions compute the floating-point remainder that results from dividing the first argument by the second argument.

The result is less than the second argument and has the same sign as the first argument. If the second argument is equal to zero, the `fmod` functions return zero.

Error Conditions

The `fmod` functions do not return an error condition.

Example

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

y = fmod (5.0, 2.0);      /* y = 1.0 */
x = fmodf (4.0, 2.0);     /* x = 0.0 */
```

See Also

div, ldiv, modf

fopen

Open a file

Synopsis

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Description

The `fopen` function initializes the data structures that are required for reading or writing to a file. The file's name is identified by `filename`, with the access type required specified by the string `mode`.

Valid selections for `mode` are specified in Table 1-36. If any other mode specification is selected then the behavior is undefined.

Table 1-36. Valid Selections for mode

mode	Selection
`r`	Open text file for reading. This operation fails if the file has not previously been created.
`w`	Open text file for writing. If the filename already exists then it will be truncated to zero length with the write starting at the beginning of the file. If the file does not already exist then it is created.
`a`	Open a text file for appending data. All data will be written to the end of the file specified.
`r+`	As `r` with the exception that the file can also be written to.
`w+`	As `w` with the exception that the file can also be read from.
`a+`	As `a` with the exception that the file can also be read from any position within the file. Data is only written to the end of the file.
`rb`	As `r` with the exception that the file is opened in binary mode.
`wb`	As `w` with the exception that the file is opened in binary mode.
`ab`	As `a` with the exception that the file is opened in binary mode.

Table 1-36. Valid Selections for mode (Cont'd)

mode	Selection
`r+b/rb+`	Open file in binary mode for both reading and writing.
`w+b/wb+`	Create or truncate to zero length a file for both reading and writing.
`a+b/ab+`	As `a+` with the exception that the file is opened in binary mode.

If the call to the `fopen` function is successful a pointer to the object controlling the stream is returned.

Error Conditions

If the `fopen` function is not successful a `NULL` pointer is returned.

Example

```
#include <stdio.h>

FILE *open_output_file(void)
{
   /* Open file for writing as binary */
   FILE *handle = fopen("output.dat", "wb");
   return handle;
}
```

See Also

fclose, fflush, freopen

fprintf

Print formatted output

Synopsis

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, /*args*/ ...);
```

Description

The `fprintf` function places output on the named output `stream`. The string pointed to by `format` specifies how the arguments are converted for output.

The format string can contain zero or more conversion specifications, each beginning with the `%` character. The conversion specification itself follows the `%` character and consists of one or more of the following sequence:

- Flag – optional characters that modifies the meaning of the conversion.
- Width – optional numeric value (or *) that specifies the minimum field width.
- Precision – optional numeric value that gives the minimum number of digits to appear.
- Length – optional modifier that specifies the size of the argument.
- Type – character that specifies the type of conversion to be applied.

The flag characters can be in any order and are optional. The valid flags are described in Table 1-37.

Table 1-37. Valid Flags for fprintf Function

Flag	Field
`-`	Left justify the result within the field. The result is right-justified by default.
+	Always begin a signed conversion with a plus or minus sign. By default only negative values will start with a sign.
`space`	Prefix a space to the result if the first character is not a sign and the + flag has not also been specified.
#	The result is converted to an alternative form depending on the type of conversion: o : If the value is not zero it is preceded with 0. x : If the value is not zero it is preceded with 0x. X : If the value is not zero it is preceded with 0X. a A e E f F: Always generate a decimal point. g G : as E except trailing zeros are not removed.
`0` (zero)	Specifies an alternative to space padding. Leading zeroes will be used as necessary to pad a field to the specified field width, the leading zeroes will follow any sign or specification of a base. The flag will be ignored if it appears with a '-' flag or if it is used in a conversion specification that uses a precision and one of the conversons `a`, `A`, `d`, `i`, `o`, `u`, `x` or `X`. The `0` flag may be used with the `a`, `A`, `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g` and `G` conversions.

If a field width is specified, the converted value is padded with spaces to the specified width if the converted value contains fewer characters than the width. Normally spaces will be used to pad the field on the left, but padding on the right will be used if the '`-`' flag has been specified. The '`0`' flag may be used as an alternative to space padding; see the description of the flag field above. The width may also be specified as a '*', which indicates that the current argument in the call to `fprintf` is an `int` that defines the value of the width. If the value is negative then it is interpreted as a '`-`' flag and a positive field width.

The optional precision value always begins with a period (.) and is followed either by an asterisk (*) or by a decimal integer. An asterisk (*) indicates that the precision is specified by an integer argument preceding the argument to be formatted. If only a period is specified, a precision of zero will be assumed. The precision value has differing effects depending on the conversion specifier being used:

- For `A`, `a` specifies the number of digits after the decimal point. If the precision is zero and the `#` flag is not specified no decimal point will be generated.
- For `d,i,o,u,x,X` specifies the minimum number of digits to appear, defaulting to 1.
- For `f,F,E,e,r,R` specifies the number of digits after the decimal point character, the default being 6. If the `#` specifier is present with a zero precision then no decimal point will be generated.
- For `g`, `G` specifies the maximum number of significant digits.
- For `s` specifies the maximum number of characters to be written.

The length modifier (Table 1-38) can optionally be used to specify the size of the argument. The length modifiers should only precede one of the `d, i, o, u, x, X, r, R` or `n` conversion specifiers unless other conversion specifiers are detailed.

Table 1-38. Length Modifiers for fprintf Function

Length	Action
`h`	The argument should be interpreted as a `short` int. If preceding the `r` or `R` conversion specifier, the argument is interpreted as `short fract` or `unsigned short fract`.
`l`	The argument should be interpreted as a `long` int. If preceding the `r` or `R` conversion specifier, the argument is interpreted as `long fract` or `unsigned long fract`

Table 1-38. Length Modifiers for fprintf Function (Cont'd)

Length	Action
`ll`	The argument should be interpreted as a `long long` int.
`L`	The argument should be interpreted as a `long double` argument. This length modifier should precede one of the `a, A, e, E, f, F, g,` or `G` conversion specifiers. Note that this length modifier is only valid if `-double-size-64` is selected. If `-double-size-32` is selected no conversion will occur, with the corresponding argument being consumed.

Table 1-39 contains definitions of the valid conversion specifiers that define the type of conversion applied.

Table 1-39. Valid Conversion Specifier Definitions for fprintf Function

Specifier	Conversion
`a, A`	floating-point, hexadecimal notation
`c`	character
`d, i`	signed decimal integer
`e, E`	floating-point, scientific notation (mantissa/exponent)
`f, F`	floating-point, decimal notation
`g, G`	convert as e, E or f, F
`n`	pointer to signed integer to which the number of characters written so far will be stored with no other output
`o`	unsigned octal
`p`	pointer to void
`r`	signed fract
`R`	unsigned fract
`s`	string of characters
`u`	unsigned integer
`x, X`	unsigned hexadecimal notation
`%`	print a `%` character with no argument conversion

The a|A conversion specifier converts to a floating-point number with the notational style [-]0xh.hhhh±d where there is one hexadecimal digit before the period. The a|A conversion specifiers always contain a minimum of one digit for the exponent.

The e|E conversion specifier converts to a floating-point number notational style [-]d.ddde±dd. The exponent always contains at least two digits. The case of the e preceding the exponent will match that of the conversion specifier.

The f|F conversion specifies to convert to decimal notation [-]d.ddd±ddd.

The g|G conversion specifier converts as e|E or f|F specifiers depending on the value being converted. If the value being converted is less than -4 or greater than or equal to the precision then e|E conversions will be used, otherwise f|F conversions will be used.

For all of the a, A, e, E, f, F, g and G specifiers an argument that represents infinity is displayed as Inf. For all of the a, A, e, E, f, F, g and G specifiers an argument that represents a NaN result is displayed as NaN.

The r|R conversion specifiers convert a fixed-point value to decimal notation [-]*d.ddd* if you are linking with the fixed-point I/O library using the -flags-link -MD__LIBIO_FX switch. Otherwise they will convert a fixed-point value to hexadecimal.

The fprintf function returns the number of characters printed.

Error Conditions

If the fprintf function is unsuccessful, a negative value is returned.

Example

```
#include <stdio.h>

void fprintf_example(void)
{
   char *str = "hello world";
   /* Output to stdout is "   +1  +1." */
   fprintf(stdout, "%+5.0f%+#5.0f\n", 1.234, 1.234);


   /* Output to stdout is "1.234 1.234000 1.23400000" */
   fprintf(stdout, "%.3f %f %.8f\n", 1.234, 1.234, 1.234);


   /* Output to stdout is "justified:
                           left:5    right:    5" */
   fprintf(stdout, "justified:\nleft:%-5dright:%5i\n", 5, 5);


   /* Output to stdout is
      "90% of test programs print hello world" */
   fprintf(stdout, "90%% of test programs print %s\n", str);


   /* Output to stdout is "0.0001 1e-05 100000 1E+06" */
   fprintf(stdout, "%g %g %G %G\n", 0.0001, 0.00001, 1e5, 1e6);
}
```

See Also

printf, snprintf, vfprintf, vprintf, vsnprintf, vsprintf

fputc

Put a character on a stream

Synopsis

```
#include <stdio.h>
int fputc(int ch, FILE *stream);
```

Description

The `fputc` function writes the argument `ch` to the output stream pointed to by `stream` and advances the file position indicator. The argument `ch` is converted to an `unsigned char` before it is written.

If the `fputc` function is successful then it will return the value that was written to the stream.

Error Conditions

If the `fputc` function is not successful `EOF` is returned.

Example

```
#include <stdio.h>

void fputc_example(FILE* fp)
{
   /* put the character 'i' to the stream pointed to by fp */
   int res = fputc('i', fp);
   if (res != 'i')
      printf("fputc failed\n");
}
```

See Also

putc

fputs

Put a string on a stream

Synopsis

```
#include <stdio.h>
int fputs(const char *string, FILE *stream);
```

Description

The `fputs` function writes the string pointed to by `string` to the output stream pointed to by `stream`. The `NULL` terminating character of the string will not be written to `stream`.

If the call to `fputs` is successful, the function returns a non-negative value.

Error Conditions

The `fputs` function will return `EOF` if a write error occurred.

Example

```
#include <stdio.h>

void fputs_example(FILE* fp)
{
   /* put the string "example" to the stream pointed to by fp */
   char *example = "example";
   int res = fputs(example, fp);
   if (res == EOF)
      printf("fputs failed\n");
}
```

See Also

puts

fread

Buffered input

Synopsis

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

Description

The `fread` function reads into an array pointed to by `ptr` up to a maximum of `n` items of data from `stream`, where each item of data is of length `size`. It stops reading data if an `EOF` or error condition is encountered while reading from `stream`, or if `n` items have been read. It advances the data pointer in `stream` by the number of characters read. It does not change the contents of `stream`.

The `fread` function returns the number of items read, this may be less than `n` if there is insufficient data on the external device to satisfy the read request. If `size` or `n` is zero, then `fread` will return zero and does not affect the state of `stream`.

When the stream has been opened as a binary stream, the Analog Devices I/O library may choose to bypass the I/O buffer and transmit data from an external device directly into the program, particularly when the buffer size (as defined by the macro `BUFSIZ` in the `stdio.h` header file, or controlled by the function `setvbuf`) is smaller than the number of characters to be transferred.

Normally, binary streams are a bit-exact mirror image of the processor's memory such that data that is written out to a binary stream can be later read back unmodified. The size of a binary file on SHARC architecture is therefore normally a multiple of 32-bit words. When the size of a file is not a multiple of four, `fread` will behave as if the file was padded out by a sufficient number of trailing null characters to bring the size of the file up to the next multiple of 32-bit words.

Error Conditions

If an error occurs, `fread` returns zero and sets the error indicator for `stream`.

Example

```
#include <stdio.h>

int buffer[100];

int fill_buffer(FILE *fp)
{
   int read_items;
   /* Read from file pointer fp into array buffer  */
   read_items = fread(&buffer, sizeof(int), 100, fp);
   if (read_items < 100) {
       if (ferror(fp))
          printf("fill_buffer failed with an I/O error\n");
       else if (feof(fp))
          printf("fill_buffer failed with EOF\n");
       else
          printf("fill_buffer only read %d items\n",read_items);
   }
   return read_items;
}
```

See Also

ferror, fgetc, fgets, fscanf

free

Deallocate memory

Synopsis

```
#include <stdlib.h>
void free (void *ptr);
```

Description

The `free` function deallocates a pointer previously allocated to a range of memory (by `calloc` or `malloc`) to the free memory heap. If the pointer was not previously allocated by `calloc`, `malloc`, `realloc`, `heap_calloc`, `heap_malloc`, or `heap_realloc`, the behavior is undefined.

The `free` function returns the allocated memory to the heap from which it was allocated.

Error Conditions

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

Example

```
#include <stdlib.h>

char *ptr;

ptr = malloc (10);      /* Allocate 10 words from heap */
free (ptr);             /* Return space to free heap   */
```

See Also

calloc, heap_calloc, heap_free, heap_lookup_name, heap_malloc, heap_realloc, malloc, realloc, set_alloc_type

freopen

Open a file using an existing file descriptor

Synopsis

```
#include <stdio.h>
FILE *freopen(const char *fname, const char *mode, FILE *stream);
```

Description

The `freopen` function opens the file specified by `fname` and associates it with the stream pointed to by `stream`. The mode argument has the same effect as described in `fopen`. (See "fopen" on page 1-151 for more information on the mode argument.)

Before opening the new file the `freopen` function will first attempt to flush the stream and close any file descriptor associated with `stream`. Failure to flush or close the file successfully is ignored. Both the error and `EOF` indicators for `stream` are cleared.

The original stream will always be closed regardless of whether the opening of the new file is successful or not.

Upon successful completion the `freopen` function returns the value of `stream`.

Error Conditions

If `freopen` is unsuccessful, a `NULL` pointer is returned.

Example

```
#include <stdio.h>

void freopen_example(FILE* fp)
{
   FILE *result;
   char *newname = "newname";


   /* reopen existing file pointer for reading file "newname" */
   result = freopen(newname, "r", fp);
   if (result == fp)
      printf("%s reopened for reading\n", newname);
   else
      printf("freopen not successful\n");
}
```

See Also

fclose, fopen

frexp

Separate fraction and exponent

Synopsis

```
#include <math.h>

float frexpf (float x, int *expptr);
double frexp (double x, int *expptr);
long double frexpd (long double x, int *expptr);
```

Description

The `frexp` functions separate a floating-point input into a normalized fraction and a (base 2) exponent. The functions return a fraction in the interval [½, 1), and store a power of 2 in the integer pointed to by the second argument. If the input is zero, then both the fraction and the exponent is set to zero.

Error Conditions

The `frexp` functions do not return an error condition.

Example

```
#include <math.h>

double y;
float x;
int exponent;

y = frexp (2.0, &exponent);       /* y = 0.5, exponent = 2 */
x = frexpf (5.0, &exponent);      /* x = 0.5, exponent = 3 */
```

See Also

modf

fscanf

Read formatted input

Synopsis

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, /* args */...);
```

Description

The `fscanf` function reads from the input file `stream`, interprets the inputs according to `format` and stores the results of the conversions (if any) in its arguments. The `format` is a string containing the control format for the input with the following arguments as pointers to the locations where the converted results are written.

The string pointed to by `format` specifies how the input is to be parsed and, possibly, converted. It may consist of whitespace characters, ordinary characters (apart from the `%` character), and conversion specifications. A sequence of whitespace characters causes `fscanf` to continue to parse the input until either there is no more input or until it finds a non-whitespace character. If the format specification contains a sequence of ordinary characters then `fscanf` will continue to read the next characters in the input stream until the input data does not match the sequence of characters in the format. At this point `fscanf` will fail, and the differing and subsequent characters in the input stream will not be read.

The `%` character in the format string introduces a conversion specification. A conversion specification has the following form:

```
% [*] [width] [length] type
```

A conversion specification always starts with the `%` character. It may optionally be followed by an asterisk (*) character, which indicates that the result of the conversion is not to be saved. In this context the asterisk character is known as the assignment-suppressing character. The optional token width represents a non-zero decimal number and specifies the maximum field width. fscanf will not read any more than width characters while performing the conversion specified by type. The length token can be used to define a length modifier.

The length modifier (Table 1-40) can be used to specify the size of the argument. The length modifiers should only precede one of the d, i, o, u, x, X, r, R or n conversion specifiers unless other conversion specifiers are detailed.

Table 1-40. Length Modifiers for fscanf Function

Length	Action
h	The argument should be interpreted as a short int. If preceding the r or R conversion specifier, the argument is interpreted as short fract or unsigned short fract.
hh	The argument should be interpreted as a char.
j	The argument should be interpreted as intmax_t or uintmax_t.
l	The argument should be interpreted as a long int. If preceding the r or R conversion specifier, the argument is interpreted as long fract or unsigned long fract.
L	The argument should be interpreted as a long double argument. This length modifier should precede one of the a, A, e, E, f, F, g, or G conversion specifiers.
t	The argument should be interpreted as ptrdiff_t.
z	The argument should be interpreted as size_t.

The hh, j, t, and z size specifiers are defined in the C99 (ISO/IEC 9899:1999) standard.

A definition of the valid conversion specifier characters that specify the type of conversion to be applied can be found in Table 1-41.

Table 1-41. Valid Conversion Specifier Definitions for fscanf Function

Specifier	Conversion
`a A e E f F g G`	floating point, optionally preceded by a sign and optionally followed by an e or E character
`c`	single character, including whitespace
`d`	signed decimal integer with optional sign
`i`	signed integer with optional sign
`n`	no input is consumed. The number of characters read so far will be written to the corresponding argument. This specifier does not affect the function result returned by `fscanf`
`o`	unsigned octal
`p`	pointer to void
`r`	signed fract with optional sign
`R`	unsigned fract
`s`	string of characters up to a whitespace character
`u`	unsigned decimal integer
`x X`	hexadecimal integer with optional sign
`[`	a non-empty sequence of characters referred to as the scanset
`%`	a single `%` character with no conversion or assignment

The `[` conversion specifier should be followed by a sequence of characters, referred to as the `scanset`, with a terminating `]` character and so will take the form `[scanset]`. The conversion specifier copies into an array which is the corresponding argument until a character that does not match any of the scanset is read. If the scanset begins with a `^` character then the scanning will match against characters not defined in the scanset. If the scanset is to include the `]` character, then this character must immediately follow the `[` character or the `^` character if specified.

Each input item is converted to a type appropriate to the conversion character, as specified in the table above. The result of the conversion is placed into the object pointed to by the next argument that has not already been the recipient of a conversion. If the suppression character has been specified then no data shall be placed into the object with the next conversion using the object to store its result.

Note that the `r` and `R` format specifiers are only supported when linking with the fixed-point I/O library using `-flags-link -MD__LIBIO_FX`.

The `fscanf` function returns the number of items successfully read.

Error Conditions

If the `fscanf` function is not successful before any conversion then `EOF` is returned.

Example

```
#include <stdio.h>

void fscanf_example(FILE *fp)
{
   short int day, month, year;
   float f1, f2, f3;
   char string[20];

   /* Scan a date with any separator, eg, 1-1-2006 or 1/1/2006 */
   fscanf (fp, "%hd%*c%hd%*c%hd", &day, &month, &year);
   /* Scan float values separated by "abc", for example
      1.234e+6abc1.234abc235.06abc                        */
   fscanf (fp, "%fabc%gabc%eabc", &f1, &f2, &f3);

   /* For input "alphabet", string will contain "a" */
   writ(fp, "%[aeiou]", string);
   /* For input "drying”, string will contain "dry" */
   fscanf (fp, "%[^aeiou]", string);
}
```

See Also

scanf, sscanf

fseek

Reposition a file position indicator in a stream

Synopsis

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
```

Description

The `fseek` function sets the file position indicator for the stream pointed to by `stream`. The position within the file is calculated by adding the offset to a position dependent on the value of `whence`. The valid values and effects for `whence` are found in Table 1-42.

Table 1-42. Valid Values and Effects for whence

whence	Effect
`SEEK_SET`	Set the position indicator to be equal to `offset` characters from the beginning of `stream`.
`SEEK_CUR`	Set the new position indicator to current position indicator for `stream` plus `offset`.
`SEEK_END`	Set the position indicator to `EOF` plus `offset`.

Using `fseek` to position a text stream is only valid if either `offset` is zero, or if `whence` is `SEEK_SET` and `offset` is a value that was previously returned by `ftell`. For binary streams, the offset is measured in addressable units of memory, which on SHARC is 32-bit words.

Positioning within a file that has been opened as a text stream is only supported by the libraries that Analog Devices supply if the lines within the file are terminated by the character sequence `\r\n`.

A successful call to `fseek` will clear the `EOF` indicator for `stream` and undoes any effects of `ungetc` on `stream`. If the stream has been opened as a update stream, then the next I/O operation may be either a read request or a write request.

Error Conditions

If the `fseek` function is unsuccessful, a non-zero value is returned.

Example

```
#include <stdio.h>

long fseek_and_ftell(FILE *fp)
{
   long offset;
   /* seek to 20 characters offset from given file pointer */
   if (fseek(fp, 20, SEEK_SET) != 0) {
      printf("fseek failed\n");
      return -1;
   }
   /* Now use ftell to get the offset value back */
   offset = ftell(fp);
   if (offset == -1)
       printf("ftell failed\n");
   if (offset == 20)
       printf("ftell and fseek work\n");
   return offset;
}
```

See Also

fflush, ftell, ungetc

fsetpos

Reposition a file pointer in a stream

Synopsis

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description

The `fsetpos` function sets the file position indicator for `stream`, using the value of the object pointed to by `pos`. The value pointed to by `pos` must be a value obtained from an earlier call to `fgetpos` on the same stream.

Positioning within a file that has been opened as a text stream is only supported by the libraries that Analog Devices supply if the lines within the file are terminated by the character sequence `\r\n`.

A successful call to `fsetpos` function clears the `EOF` indicator for `stream` and undoes any effects of `ungetc` on the same stream.

The `fsetpos` function returns zero if it is successful.

Error Conditions

If the `fsetpos` function is unsuccessful, the function returns a non-zero value.

Example

Refer to "fgetpos" on page 1-145 for an example.

See Also

fgetpos, ftell, rewind, ungetc

ftell

Obtain current file position

Synopsis

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Description

The `ftell` function obtains the current position for a file identified by `stream`.

If `stream` is a text stream, then the information in the position indicator is unspecified information, usable by `fseek` for determining the file position indicator at the time of the `ftell` call.

If `stream` is a binary stream, then `ftell` returns the current position as an offset from the start of the file. As binary streams are normally bit-exact images of the processor's memory, the offset returned is in addressable units of memory that, on a SHARC processor, is 32-bit words.

Positioning within a file that has been opened as a text stream is only supported by the libraries that Analog Devices supply if the lines within the file are terminated by the character sequence `\r\n`.

If successful, the `ftell` function returns the current value of the file position indicator on the stream.

Error Conditions

If the `ftell` function is unsuccessful, a value of -1 is returned.

Example

See "fseek" on page 1-173 for an example.

See Also

fseek

fwrite

Buffered output

Synopsis

```
#include <stdio.h>

size_t fwrite(const void *ptr, size_t size, size_t n,
              FILE *stream);
```

Description

The `fwrite` function writes to the output `stream` up to `n` items of data from the array pointed by `ptr`. An item of data is defined as a sequence of characters of size `size`. The write will complete once `n` items of data have been written to the stream. The file position indicator for `stream` is advanced by the number of characters successfully written.

When the stream has been opened as a binary stream, the Analog Devices I/O library may choose to bypass the I/O buffer and transmit data from the program directly to the external device, particularly when the buffer size (as defined by the macro `BUFSIZ` in the `stdio.h` header file, or controlled by the function `setvbuf`) is smaller than the number of characters to be transferred.

If successful then the `fwrite` function will return the number of items written.

Error Conditions

If the `fwrite` function is unsuccessful, it will return the number of elements successfully written which will be less than `n`.

Example

```
#include <stdio.h>
char* message="some text";
void write_text_to_file(void)
{
   /* Open "file.txt" for writing */
   FILE* fp = fopen("file.txt", "w");
   int res, message_len = strlen(message);
   if (!fp) {
      printf("fopen was not successful\n");
      return;
   }
   res = fwrite(message, sizeof(char), message_len, fp);
   if (res != message_len)
      printf("fwrite was not successful\n");
}
```

See Also

fread

fxbits

Bitwise integer to fixed-point to conversion

Synopsis

```
#include <stdfix.h>

short fract hrbits(int_hr_t b);
fract rbits(int_r_t b);
long fract lrbits(int_lr_t b);
unsigned short fract uhrbits(uint_uhr_t b);
unsigned fract urbits(uint_ur_t b);
unsigned long fract ulrbits(uint_ulr_t b);
```

Description

Given an integer operand, the `fxbits` family of functions return the integer value divided by 2^F, where `F` is the number of fractional bits in the result fixed-point type. This is equivalent to the bit-pattern of the integer value held in a fixed-point type.

Error Conditions

The `fxbits` family of functions do not return an error condition. If the input integer value does not fit in the number of bits of the fixed-point result type, the result is saturated to the largest or smallest fixed-point value.

Example

```
#include <stdfix.h>
unsigned long fract ulr;
ulr = ulrbits(0x20000000);            /* ulr == 0.125ulr */
```

See Also

bitsfx

fxdivi

Division of integer by integer to give fixed-point result

Synopsis

```
#include <stdfix.h>

fract rdivi(int numer, int denom);
long fract lrdivi(long int numer, long int denom);
unsigned fract urdivi(unsigned int numer, unsigned int denom);
unsigned long fract ulrdivi(unsigned long int numer,
                            unsigned long int denom);
```

Description

Given an integer numerator and denominator, the `fxdivi` family of functions computes the quotient and returns the closest fixed-point value to the result.

Error Conditions

The `fxdivi` family of functions have undefined behavior if the denominator is zero.

Example

```
#include <stdfix.h>
unsigned long fract ulquo;
ulquo = ulrdivi(1, 8);        /* ulquo == 0.125ulr */
```

See Also

div, divifx, idivfx, ldiv, lldiv

getc

Get a character from a stream

Synopsis

```
#include <stdio.h>
int getc(FILE *stream);
```

Description

The `getc` function is equivalent to `fgetc`. The `getc` function obtains the next character from the input stream pointed to by `stream`, converts it from an `unsigned char` to an `int` and advances the file position indicator for the stream.

Upon successful completion the `getc` function will return the next character from the input stream pointed to by `stream`.

Error Conditions

If the `getc` function is unsuccessful, `EOF` is returned.

Example

```
#include <stdio.h>

char use_getc(FILE *fp)
{
   char ch;
   if ((ch = getc(fp)) == EOF) {
      printf("Read End-of-file\n");
      return (char)-1;
   } else {
      return ch;
   }
}
```

See Also

fgetc

getchar

Get a character from stdin

Synopsis

```
#include <stdio.h>
int getchar(void);
```

Description

The `getchar` function is functionally the same as calling the `getc` function with `stdin` as its argument. A call to `getchar` will return the next single character from the standard input stream. The `getchar` function also advances the standard input's current position indicator.

Error Conditions

If the `getchar` function is unsuccessful, `EOF` is returned.

Example

```
#include <stdio.h>

char use_getchar(void)
{
   char ch;
   if ((ch = getchar()) == EOF) {
      printf("getchar() failed\n");
      return (char)-1;
   } else {
      return ch;
   }
}
```

See Also

getc

getenv

Get string definition from operating system

Synopsis

```
#include <stdlib.h>
char *getenv (const char *name);
```

Description

The `getenv` function polls the operating system to see if a string is defined. There is no default operating system for the SHARC processors, so `getenv` always returns NULL.

Error Conditions

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

Example

```
#include <stdlib.h>

char *ptr;

ptr = getenv ("ADI_DSP");      /* ptr = NULL */
```

See Also

system

gets

Get a string from a stream

Synopsis

```
#include <stdio.h>
char *gets(char *s);
```

Description

The `gets` function reads characters from the standard input stream into the array pointed to by `s`. The read terminates when a `NEWLINE` character is read, with the `NEWLINE` character being replaced by a null character in the array pointed to by `s`. The read will also halt if `EOF` is encountered.

The array pointed to by `s` must be of equal or greater length of the input line being read. If this is not the case, the behavior is undefined. If `EOF` is encountered without any characters being read, then a `NULL` pointer is returned.

Error Conditions

If the `gets` function is not successful and a read error occurs, then a `NULL` pointer is returned.

Example

```
#include <stdio.h>

void fill_buffer(char *buffer)
{
   if (gets(buffer) == NULL)
      printf("gets failed\n");
   else
      printf("gets read %s\n", buffer);
}
```

See Also

fgetc, fgets, fread, fscanf

gmtime

Convert calendar time into broken-down time as UTC

Synopsis

```
#include <time.h>
struct tm *gmtime(const time_t *t);
```

Description

The `gmtime` function converts a pointer to a calendar time into a broken-down time in terms of Coordinated Universal Time (UTC). A broken-down time is a structured variable, which is described in "time.h" on page 1-35.

The broken-down time is returned by `gmtime` as a pointer to static memory, which may be overwritten by a subsequent call to either `gmtime`, or to `localtime`.

Error Conditions

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

Example

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

time_t cal_time;
struct tm *tm_ptr;

cal_time = time(NULL);
if (cal_time != (time_t) -1) {
    tm_ptr = gmtime(&cal_time);
    printf("The year is %4d\n",1900 + (tm_ptr->tm_year));
}
```

See Also

localtime, mktime, time

heap_calloc

Allocate and initialize memory in a heap

Synopsis

```
#include <stdlib.h>
void *heap_calloc(int heap_index, size_t nelem, size_t size);
```

Description

The `heap_calloc` function is an Analog Devices extension to the ANSI standard.

The `heap_calloc` function allocates from the heap identified by `heap_index`, an array containing `nelem` elements of `size`, and stores zeros in all the elements of the array. If successful, it returns a pointer to this array; otherwise, it returns a null pointer. You can safely convert the return value to an object pointer of any type whose size is not greater than `size`. The memory may be deallocated with the `free` or `heap_free` function.

For more information on creating multiple run-time heaps, see Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, section "Using Multiple Heaps".

Error Conditions

The `heap_calloc` function returns the null pointer if unable to allocate the requested memory.

Example

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

int main()
{
   char *buf;
   int index;
   /* Obtain the heap index for "seg_hp2" */
   index = heap_lookup_name("seg_hp2");
   if (index < 0) {
      printf("Heap with name seg_hp2 not found\n");
      return 1;
   }

   /* Allocate memory for 128 characters from seg_hp2 */
   buf = (char *)heap_calloc(index,128,sizeof(char));
   if (buf != 0) {
      printf("Allocated space from %p\n", buf);
      free(buf);   /* free can be used to release the memory */
   } else {
      printf("Unable to allocate from seg_hp2\n");
   }
   return 0;
}
```

See Also

calloc, free, heap_free, heap_lookup_name, heap_malloc, heap_realloc, malloc, realloc, set_alloc_type

heap_free

Return memory to a heap

Synopsis

```
#include <stdlib.h>
void heap_free(int heap_index, void *ptr);
```

Description

The `heap_free` function is an Analog Devices extension to the ANSI standard.

If `ptr` is not a null pointer, the `heap_free` function deallocates the object whose address is `ptr`; otherwise, it does nothing. The argument `heap_index` must be the index of the heap from which the object pointed to by `ptr` was originally allocated. If the object was not allocated from the specified heap, then the behavior is undefined.

The `heap_free` function is somewhat faster than `free`, but `free` must be used if the heap from which the object was allocated is not known with certainty.

For more information on creating multiple run-time heaps, see Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, section "Using Multiple Heaps".

Error Conditions

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

Example

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

int main()
{
   char *buf;
   int index;
   /* Obtain the heap index for "seg_hp2" */
   index = heap_lookup_name("seg_hp2");
   if (index < 0) {
      printf("Heap with name seg_hp2 not found\n");
      return 1;
   }

   /* Allocate memory for 128 characters from seg_hp2 */
   buf = (char *)heap_calloc(index,128,sizeof(char));
   if (buf != 0) {
      printf("Allocated space from %p\n", buf);
      heap_free(index, buf);      /* heap_free can be used */
                                  /* to release the memory */
   } else {
      printf("Unable to allocate from seg_hp2\n");
   }
   return 0;
}
```

See Also

calloc, free, heap_calloc, heap_lookup_name, heap_malloc, heap_realloc, malloc, realloc, set_alloc_type

heap_install

Sets up a heap at run-time

Synopsis

```
#include <stdlib.h>

int heap_install(void *base, size_t length, int userid,
                 int pmdm);
```

Description

The `heap_install` function is an Analog Devices extension to the ANSI standard.

The `heap_install` function sets up a memory heap (`base`) with a size specified by `length` at run-time. The dynamic heap is identified by the `userid` and resides in either DM if `pmdm` has a value of `-1` or PM memory if `pmdm` has a value of `1`.

On successful initialization, `heap_install()` returns the heap index allocated for the newly installed heap. If the operation is unsuccessful, then `heap_install()` returns -1.

Once the dynamic heap is initialized, heap space can be claimed using the `heap_malloc` routine and associated heap management routines.

Note that the `heap_lookup_name` function does not work with a heap dynamically initialized by `heap_install()`. The `heap_lookup_name` function only works with statically initialized heaps.

Error Conditions

The `heap_install` function returns -1 if initialization was unsuccessful. This may be because there is not enough space available in the `__heaps` table, or if a heap with the specified `userid` already exists.

Example

```
<< Linker Description File >>

MEMORY
{
      ..
      seg_runtime_dm { TYPE(DM RAM)
          START(0x0005b000) END(0x0005dfff) WIDTH(32) }
      ..
   }

PROCESSOR p0
{
      ..

   SECTIONS
   {
      ..

      seg_runtime_dm
      {
          _start_of_seg_runtime_dm = .;
      } > seg_runtime_dm
    }
}

 << C Source File >>

   #include <stdlib.h>
   extern int __start_of_seg_runtime_dm;

   #define DM_MEM     -1
   #define ADDR_DM    &__start_of_seg_runtime_dm

   int main()
```

```
{
   int i;
   int index;
   int *x;

   index = heap_install(
           (void *)ADDR_DM, 100, 3, DM_MEM);
   if (index != -1)
      x = heap_malloc(index, 90*sizeof(int));

   if (x) {
      for (i = 0; i < 90; i++)
          x[i] = i;
   }

   return 0;
}
```

See Also

heap_lookup_name, heap_malloc

heap_lookup_name

Obtain primary heap identifier

Synopsis

```
#include <stdlib.h>
int heap_lookup_name(char *user_id);
```

Description

The `heap_lookup_name` function is an Analog Devices extension to the ANSI standard.

The `heap_lookup_name` function returns the primary heap identifier of the heap with user identifier `user_id`, if there is such a heap; otherwise, -1 is returned. The primary heap identifier is the index of the heap descriptor record in the heap descriptor table. The user identifier for a heap is determined by a field in the heap descriptor record. The default heap always has user identifier 0.

For more information on multiple run-time heaps, see Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, section "Using Multiple Heaps".

Error Conditions

The function returns -1 if the specified user identifier was not found, otherwise it returns the primary heap identifier of the specified heap.

Example

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

void func2(int pm * b);

func()
```

```
{
   int pm * x;
   int loop, pm_heapID;

   pm_heapID = heap_lookup_name("seg_heaq");

   if (pm_heapID < 0) {
      printf("Lookup failed\n");
      return 1;
   }

   x = (int pm *)heap_malloc(pm_heapID, 1000);
                              // Get 1K words of PM heap space
   if (x == NULL) {
      printf("heap_malloc failed\n");
      return 1;
   }

   for (loop = 0; loop < 1000; loop++)
       x[loop] = loop;

   func2(x);                  // Do something with x
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_malloc, heap_realloc, malloc, realloc, set_alloc_type

heap_malloc

Allocate memory from a heap

Synopsis

```
#include <stdlib.h>
void *heap_malloc(int heap_index, size_t size);
```

Description

The `heap_malloc` function is an Analog Devices extension to the ANSI standard.

The `heap_malloc` function allocates an object of `size` from the heap identified by `heap_index`. It returns the address of the object if successful; otherwise, it returns a null pointer. You can safely convert the return value to an object pointer of any type whose size is not greater than `size`.

The block of memory is uninitialized. The memory may be deallocated with the `free` or `heap_free` function.

For more information on creating multiple run-time heaps, see Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, section "Using Multiple Heaps".

Error Conditions

The `heap_malloc` function returns the null pointer if unable to allocate the requested memory.

Example

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

int main()
```

```
{
    char *buf;
    int index;
    /* Obtain the heap index for "seg_hp2" */
    index = heap_lookup_name("seg_hp2");
    if (index < 0) {
       printf("Heap with name seg_hp2 not found\n");
       return 1;
    }

    /* Allocate memory for 128 characters from seg_hp2 */
    buf = (char *)heap_malloc(index,128);
    if (buf != 0) {
       printf("Allocated space from %p\n", buf);
       free(buf);    /* free can be used to release the memory */
    } else {
       printf("Unable to allocate from seg_hp2\n");
    }
    return 0;
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_lookup_name, heap_realloc, malloc, realloc, set_alloc_type

heap_realloc

Change memory allocation from a heap

Synopsis

```
#include <stdlib.h>
void *heap_realloc(int heap_index, void *ptr, size_t size);
```

Description

The heap_realloc function is an Analog Devices extension to the ANSI standard.

The heap_realloc function changes the size of a previously allocated block of memory. The argument heap_index specifies the heap on which the object referenced by ptr is stored. The new size of the object is specified by the argument size. The modified object will contain the values of the old object up to minimum(original size, new size), while for (new size > old size) any data beyond the original size will be indeterminate.

If the function successfully reallocated the object, then it will return a pointer to the updated object. You can safely convert the return value to an object pointer of any type whose size is not greater than size in length. The behavior of the function is undefined if the object has not been allocated from the heap specified by heap_index, or if it has already been freed.

If ptr is a null pointer, then heap_realloc behaves the same as heap_malloc and the block of memory returned will be uninitialized.

If ptr is not a null pointer, and if size is zero, then heap_realloc behaves the same as heap_free.

The memory reallocated may be deallocated with the free or heap_free function.

For more information on creating multiple run-time heaps, see Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, section “Using Multiple Heaps”.

Error Conditions

The `heap_realloc` function returns the null pointer if unable to allocate the requested memory; the original memory associated with `ptr` will be unchanged and will still be available.

Example

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main()
{
   int index,ok,prev;
   char *buf,*upd;

/* Obtain the heap index for the user identifier 2 */
   index = heap_lookup_name("seg_hp2");
   if (index < 0) {
      printf("Heap with name seg_hp2 not found\n");
      return 1;
   }
   /* Allocate memory for 128 characters from seg_hp2          */
   buf = (char *)heap_malloc(index,128);
   if (buf != 0) {
       strcpy(buf,”hello”);
                                /* Change allocated size to 256 */
       upd = (char *)heap_realloc(index,buf,256);
       if (upd != 0) {
           printf("reallocated string for %s\n",upd);
```

```
            heap_free(index,upd);          /* Return to seg_hp2 */
        } else {
            free(buf);       /* free can be used to release buf */
        }
    } else {
        printf("Unable to allocate from seg_hp2\n");
    }
    return 0;
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_lookup_name, heap_malloc, malloc, realloc, set_alloc_type

heap_switch

Change the default heap at run-time

Synopsis

```
#include <stdlib.h>
int heap_switch (int heapid);
```

Description

The `heap_switch` function changes the default heap (as used by heap allocation functions `malloc`, `calloc`, `realloc` and `free`). The function returns the `heapid` of the previous default heap.

For more information on creating multiple run-time heaps, see Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, section “Using Multiple Heaps”.

The `heap_switch` function is not available in multithreaded environments.

Error Conditions

The `heap_switch` function reports no error conditions.

Example

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

#define HEAP1_USERID  1
#define HEAP1_SIZE    1024

#define DM_MEM    -1
#define PM_MEM    1
```

```
int heap1[HEAP1_SIZE];
int heap1_id;

char *pbuf;

/* Initialize */

heap1_id = heap_install (heap1, sizeof(heap1), HEAP1_USERID,
DM_MEM);

/* Make heap1 the default heap */

heap_switch (heap1_id);

/* Allocate a buffer from heap1 */

pbuf = malloc (32);
if (pbuf == NULL) {
     printf ("Unable to allocate buffer\n");
     exit (EXIT_FAILURE);
}  else {
     printf("Allocated buffer from heap1 at %p\n", pbuf);
}
```

See Also

calloc, free, malloc, realloc

idivfx

Division of fixed-point by fixed-point to give integer result

Synopsis

```
#include <stdfix.h>

int idivi(fract numer, fract denom);
long int idivlr(long fract numer, long fract denom);

unsigned int idivur(unsigned fract numer, unsigned fract denom);
unsigned long int idivulr(unsigned long fract numer,
                          unsigned long fract denom);
```

Description

Given a fixed-point numerator and denominator, the `idiv`*`fx`* family of functions computes the quotient and returns the closest integer value to the result.

Error Conditions

The `idiv`*`fx`* family of functions have undefined behavior if the denominator is zero.

Example

```
#include <stdfix.h>
unsigned long int ulquo;
ulquo = idivulr(0.5ulr, 0.125ulr);        /* ulquo == 4 */
```

See Also

div, divifx, fxdivi, ldiv, lldiv

interrupt

Define interrupt handling

Synopsis

```
#include <signal.h>

void (*interrupt (int sig, void(*func)(int))) (int);
void (*interruptnsm (int sig, void(*func)(int))) (int);
void (*interruptf (int sig, void(*func)(int))) (int);
void (*interruptfnsm (int sig, void(*func)(int))) (int);
void (*interrupts (int sig, void(*func)(int))) (int);
void (*interruptsnsm (int sig, void(*func)(int))) (int);
void (*interruptcb (int sig, void(*func)(int))) (int);
void (*interruptcbnsm (int sig, void(*func)(int))) (int);
void (*ininterruptss int sig, void(*func)(int))) (int);
void (*interruptssnsm int sig, void(*func)(int))) (int);
```

Description

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 `sig`; the `signal` function executes the function only once. The `func` argument must be one of the following that are listed in Table 1-43. The `interrupt` function causes the receipt of the signal number `sig` to be handled in one of the following ways found in Table 1-43.

Table 1-43. Interrupt Handling

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.

The differences between the functions `interrupt`, `interruptf`, `interrupts`, `interruptcb`, `interruptnsm`, `interruptfnsm`, `interruptsnsm`, `interruptcbnsm`, `interruptss`, and `interruptssnsm` are discussed under the “Support for Interrupts” section in Chapter 1 of the *VisualDSP++ C/C++ Compiler Manual for SHARC Processors*.

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 (SIG_IRQ2, irq2_handler);
/* enable interrupt 2 whose handling routine is pointed to by
irq2_handler */

interrupt (SIG_IRQ2, SIG_IGN);
/* disable interrupt 2 */
```

See Also

raise, signal

isalnum

Detect alphanumeric character

Synopsis

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

Description

The `isalnum` function determines if the argument is an alphanumeric character (`A-Z`, `a-z`, or `0-9`). If the argument is not alphanumeric, the `isalnum` function returns a zero. If the argument is alphanumeric, `isalnum` returns a non-zero value.

Error Conditions

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

Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%3s", isalnum (ch) ? "alphanumeric" : "");
    putchar ('\n');
}
```

See Also

isalpha, isdigit

isalpha

Detect alphabetic character

Synopsis

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

Description

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

isalnum, isdigit

iscntrl

Detect control character

Synopsis

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

Description

The `iscntrl` function determines if the argument is a control character (`0x00-0x1F` or `0x7F`). If the argument is not a control character, `iscntrl` returns a zero. If the argument is a control character, `iscntrl` returns a non-zero value.

Error Conditions

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

Example

```
#include <ctype.h>

int ch;

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

See Also

isalnum, isgraph

isdigit

Detect decimal digit

Synopsis

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

Description

The `isdigit` function determines if the argument `c` is a decimal digit (`0-9`). If the argument is not a digit, `isdigit` returns a zero. If the argument is a digit, `isdigit` returns a non-zero value.

Error Conditions

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

Example

```
#include <ctype.h>

int ch;

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

See Also

isalnum, isalpha, isdigit

isgraph

Detect printable character, not including white space

Synopsis

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

Description

The `isgraph` function determines if the argument is a printable character, not including a white space (`0x21-0x7e`). If the argument is not a printable character, `isgraph` returns a zero. If the argument is a printable character, `isgraph` returns a non-zero value.

Error Conditions

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

Example

```
#include <ctype.h>

int ch;

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

See Also

isalnum, iscntrl, isprint

isinf

Test for infinity

Synopsis

```
#include <math.h>

int isinff(float x);
int isinf(double x);
int isinfd(long double x);
```

Description

The `isinf` function is an Analog Devices extension to the ANSI standard.

The `isinf` functions return a zero if the argument `x` is not set to the IEEE constant for `+Infinity` or `-Infinity`; otherwise, the functions return a non-zero value.

Error Conditions

The `isinf` functions do not return or set any error conditions.

Example

```
#include <math.h>

static long val[5] = {
              0x7F7FFFFF,  /* FLT_MAX */
              0x7F800000,  /*  Inf    */
              0xFF800000,  /* -Inf    */
              0x7F808080,  /*  NaN    */
              0xFF808080,  /*  NaN    */
};
```

```
float *pval = (float *)(&val);
int m;

m = isinf (pval[0]);    /* m set to zero     */
m = isinf (pval[1]);    /* m set to non-zero */
m = isinf (pval[2]);    /* m set to non-zero */
m = isinf (pval[3]);    /* m set to zero     */
m = isinf (pval[4]);    /* m set to zero     */
```

See Also

isnan

islower

Detect lowercase character

Synopsis

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

Description

The `islower` function determines if the argument is a lowercase character (`a-z`). If the argument is not lowercase, `islower` returns a zero. If the argument is lowercase, `islower` returns a non-zero value.

Error Conditions

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

Example

```
#include <ctype.h>

int ch;

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

See Also

isalpha, isupper

isnan

Test for Not a Number (NaN)

Synopsis

```
#include <math.h>

int isnanf(float x);
int isnan(double x);
int isnand(long double x);
```

Description

The `isnan` function is an Analog Devices extension to the ANSI standard.

The `isnan` functions return a zero if the argument `x` is not set to an IEEE `NaN` (Not a Number); otherwise, the functions return a non-zero value.

Error Conditions

The `isnan` functions do not return or set any error conditions.

Example

```
#include <math.h>

static long val[5] = {
              0x7F7FFFFF,  /* FLT_MAX */
              0x7F800000,  /*  Inf    */
              0xFF800000,  /* -Inf    */
              0x7F808080,  /*  NaN    */
              0xFF808080,  /*  NaN    */
};
```

```
float *pval = (float *)(&val);
int m;

m = isnanf (pval[0]);   /* m set to zero     */
m = isnanf (pval[1]);   /* m set to zero     */
m = isnanf (pval[2]);   /* m set to zero     */
m = isnanf (pval[3]);   /* m set to non-zero */
m = isnanf (pval[4]);   /* m set to non-zero */
```

See Also

isinf

isprint

Detect printable character

Synopsis

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

Description

The `isprint` function determines if the argument is a printable character (`0x20-0x7E`). If the argument is not a printable character, `isprint` returns a zero. If the argument is a printable character, `isprint` returns a non-zero value.

Error Conditions

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

Example

```
#include <ctype.h>

int ch;

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%3s", isprint (ch) ? "printable" : "");
    putchar ('\n');
}
```

See Also

isgraph, isspace

ispunct

Detect punctuation character

Synopsis

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

Description

The `ispunct` function determines if the argument is a punctuation character. If the argument is not a punctuation character, `ispunct` returns a zero. If the argument is a punctuation character, `ispunct` returns a non-zero value.

Error Conditions

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

Example

```
#include <ctype.h>

int ch;

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%3s", ispunct (ch) ? "punctuation" : "");
    putchar ('\n');
}
```

See Also

isalnum

isspace

Detect whitespace character

Synopsis

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

Description

The `isspace` function determines if the argument is a blank whitespace character (`0x09-0x0D` or `0x20`). This includes space (` `), form feed (`\f`), new line (`\n`), carriage return (`\r`), horizontal tab (`\t`) and vertical tab (`\v`).

If the argument is not a blank whitespace character, `isspace` returns a zero. If the argument is a blank whitespace character, `isspace` returns a non-zero value.

Error Conditions

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

Example

```
#include <ctype.h>

int ch;

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

See Also

iscntrl, isgraph

isupper

Detect uppercase character

Synopsis

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

Description

The `isupper` function determines if the argument is an uppercase character (`A-Z`). If the argument is not an uppercase character, `isupper` returns a zero. If the argument is an uppercase character, `isupper` returns a non-zero value.

Error Conditions

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

Example

```
#include <ctype.h>

int ch;

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

See Also

isalpha, islower

isxdigit

Detect hexadecimal digit

Synopsis

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

Description

The `isxdigit` function determines if the argument is a hexadecimal digit character (`A-F`, `a-f`, or `0-9`). If the argument is not a hexadecimal digit, `isxdigit` returns a zero. If the argument is a hexadecimal digit, `isxdigit` returns a non-zero value.

Error Conditions

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

Example

```
#include <ctype.h>

int ch;

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

See Also

isalnum, isdigit

labs

Absolute value

Synopsis

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

Description

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

Note that `labs (LONG_MIN) == LONG_MIN`.

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, absfx, fabs, llabs

lavg

Mean of two values

Synopsis

```
#include <stdlib.h>
long int lavg (long int value1, long int value2);
```

Description

The `lavg` function is an Analog Devices extension to the ANSI standard.

The `lavg` function adds two arguments and divides the result by two. The `lavg` function is a built-in function which is implemented with an `Rn=(Rx+Ry)/2` instruction.

Error Conditions

The `lavg` function does not return an error code.

Example

```
#include <stdlib.h>

long int i;
i = lavg (10, 8);          /* returns 9 */
```

See Also

abs, avg, llavg

lclip

Clip

Synopsis

```
#include <stdlib.h>
long int lclip (long int value1, long int value2);
```

Description

The `lclip` function is an Analog Devices extension to the ANSI standard.

The `lclip` function returns the first argument if its absolute value is less than the absolute value of the second argument; otherwise it returns the absolute value of its second argument if the first is positive, or minus the absolute value if the first argument is negative. The `lclip` function is a built-in function which is implemented with an `Rn = CLIP Rx BY Ry` instruction.

Error Conditions

The `lclip` function does not return an error code.

Example

```
#include <stdlib.h>

long int i;

i = lclip (10, 8);      /* returns 8 */
i = lclip (8, 10);      /* returns 8 */
i = lclip (-10, 8);     /* returns -8 */
```

See Also

clip, fclip, llclip

lcount_ones

Count one bits in word

Synopsis

```
#include <stdlib.h>
int lcount_ones (long int value);
```

Description

The `lcount_ones` function is an Analog Devices extension to the ANSI standard.

The `lcount_ones` function returns the number of one bits in its argument.

Error Conditions

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

Example

```
#include <stdlib.h>

long int flags1 = 4095;
long int flags2 = 4096;
int cnt1;
int cnt2;

cnt1 = lcount_ones (flags1);    /* returns 12 */
cnt2 = lcount_ones (flags2);    /* returns 1  */
```

See Also

count_ones, llcount_ones

ldexp

Multiply by power of 2

Synopsis

```
#include <math.h>

float ldexpf (float x, int n);
double ldexp (double x, int n);
long double ldexpd (long double x, int n);
```

Description

The `ldexp` functions return the value of the floating-point argument multiplied by 2^n. These functions add the value of `n` to the exponent of `x`.

Error Conditions

If the result overflows, the `ldexp` functions return `HUGE_VAL` with the proper sign. If the result underflows, a zero is returned.

Example

```
#include <math.h>

double y;
float x;

y = ldexp (0.5, 2);       /* y = 2.0  */
x = ldexpf (1.0, 2);      /* x = 5.0  */
```

See Also

exp, pow

ldiv

Long division

Synopsis

```
#include <stdlib.h>
ldiv_t ldiv (long int numer, long int denom);
```

Description

The `ldiv` function divides `numer` by `denom`, and returns a structure of type `ldiv_t`. The type `ldiv_t` is defined as:

```
typedef struct {
    long int quot;
    long int rem;
} ldiv_t;
```

where `quot` is the quotient of the division and `rem` is the remainder, such that if result is of type `ldiv_t`, then
`result.quot * denom + result.rem == numer`

Error Conditions

If `denom` is zero, the behavior of the `ldiv` function is undefined.

Example

```
#include <stdlib.h>

ldiv_t result;

result = ldiv (7L, 2L);    /* result.quot = 3, result.rem = 1 */
```

See Also

div, divifx, fmod, fxdivi, idivfx, lldiv

llabs

Absolute value

Synopsis

```
#include <stdlib.h>
long long llabs (long long j);
```

Description

The `llabs` function returns the absolute value of its integer argument.

Note that `llabs (LLONG_MIN) == LLONG_MIN`.

Error Conditions

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

Example

```
#include <stdlib.h>

long long j;

j = llabs (-27081970LL);      /* j = 27081970 */
```

See Also

abs, absfx, fabs, labs

llavg

Mean of two values

Synopsis

```
#include <stdlib.h>
long long llavg (long long value1, long long value2);
```

Description

The `llavg` function is an Analog Devices extension to the ANSI standard.

The `llavg` function returns the average of the two arguments `value1` and `value2`.

Error Conditions

The `llavg` function does not return an error code.

Example

```
#include <stdlib.h>

long long i;
i = llavg (10LL, 8LL);          /* returns 9 */
```

See Also

abs, avg, lavg

llclip

Clip

Synopsis

```
#include <stdlib.h>
long long llclip (long long value1, long long value2);
```

Description

The `llclip` function is an Analog Devices extension to the ANSI standard.

The `llclip` function returns the first argument if its absolute value is less than the absolute value of the second argument; otherwise it returns the absolute value of its second argument if the first is positive, or minus the absolute value if the first argument is negative.

Error Conditions

The `llclip` function does not return an error code.

Example

```
#include <stdlib.h>

long long i;

i = llclip (10LL, 8LL);      /* returns 8 */
i = llclip (8LL, 10LL);      /* returns 8 */
i = llclip (-10LL, 8LL);     /* returns -8 */
```

See Also

clip, fclip, lclip

llcount_ones

Count one bits in long long

Synopsis

```
#include <stdlib.h>
int llcount_ones (long long value);
```

Description

The `llcount_ones` function is an Analog Devices extension to the ANSI standard.

The `llcount_ones` function returns the number of one bits in its argument.

Error Conditions

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

Example

```
#include <stdlib.h>

long long flags1 = 4095LL;
long long flags2 = 4096LL;
int cnt1;
int cnt2;

cnt1 = llcount_ones (flags1);    /* returns 12 */
cnt2 = llcount_ones (flags2);    /* returns 1  */
```

See Also

count_ones, lcount_ones

lldiv

Long long division

Synopsis

```
#include <stdlib.h>
lldiv_t lldiv (long long numer, long long denom);
```

Description

The `lldiv` function divides `numer` by `denom`, and returns a structure of type `lldiv_t`. The type `lldiv_t` is defined as:

```
typedef struct {
    long long quot;
    long long rem;
} lldiv_t;
```

where `quot` is the quotient of the division and `rem` is the remainder, such that if result is of type `lldiv_t`, then:

```
result.quot * denom + result.rem == numer
```

Error Conditions

If `denom` is zero, the behavior of the `lldiv` function is undefined.

Example

```
#include <stdlib.h>

lldiv_t result;

result = lldiv (7LL, 2LL);  /* result.quot = 3, result.rem = 1 */
```

See Also

div, divifx, fmod, fxdivi, idivfx, ldiv

llmax

Long long maximum

Synopsis

```
#include <stdlib.h>
long long llmax (long long value1, long long value2);
```

Description

The `llmax` function is an Analog Devices extension to the ANSI standard.

The `llmax` function returns the larger of its two arguments.

Error Conditions

The `llmax` function does not return an error code.

Example

```
#include <stdlib.h>

long long i;

i = llmax (10LL, 8LL);     /* returns 10 */
```

See Also

fmax, fmin, llmin, lmax, lmin, max, min

llmin

Long long minimum

Synopsis

```
#include <stdlib.h>
long long llmin (long long value1, long long value2);
```

Description

The `llmin` function is an Analog Devices extension to the ANSI standard.

The `llmin` function returns the smaller of its two arguments.

Error Conditions

The `llmin` function does not return an error code.

Example

```
#include <stdlib.h>

long long i;

i = llmin (10LL, 8LL);     /* returns 8 */
```

See Also

fmax, fmin, llmax, lmax, lmin, max, min

lmax

Long maximum

Synopsis

```
#include <stdlib.h>
long int lmax (long int value1, long int value2);
```

Description

The `lmax` function is an Analog Devices extension to the ANSI standard.

The `lmax` function returns the larger of its two arguments. The `lmax` function is a built-in function which is implemented with an `Rn = MAX(Rx,Ry)` instruction.

Error Conditions

The `lmax` function does not return an error code.

Example

```
#include <stdlib.h>

long int i;

i = lmax (10L, 8L);    /* returns 10 */
```

See Also

fmax, fmin, llmax, llmin, lmin, max, min

lmin

Long minimum

Synopsis

```
#include <stdlib.h>
long int lmin (long int value1, long int value2);
```

Description

The `lmin` function is an Analog Devices extension to the ANSI standard.

The `lmin` function returns the smaller of its two arguments. The `lmin` function is a built-in function which is implemented with an `Rn = MIN(Rx,Ry)` instruction.

Error Conditions

The `lmin` function does not return an error code.

Example

```
#include <stdlib.h>

long int i;

i = lmin (10L, 8L);     /* returns 8 */
```

See Also

fmax, fmin, lmax, llmax, llmin, max, min

localeconv

Get pointer for formatting to current locale

Synopsis

```
#include <locale.h>
struct lconv *localeconv (void);
```

Description

The `localeconv` function returns a pointer to an object of type `struct lconv`. This pointer is used to set the components of the object with values used in formatting numeric quantities in the current locale.

With the exception of `decimal_point`, those members of the structure with type `char*` may use " " to indicate that a value is not available. Expected values are strings. Those members with type `char` may use `CHAR_MAX` to indicate that a value is not available. Expected values are non-negative numbers.

The program may not alter the structure pointed to by the return value but subsequent calls to `localeconv` may do so. Also, calls to `setlocale` with the category arguments of `LC_ALL`, `LC_MONETARY` and `LC_NUMERIC` may overwrite the structure.

Table 1-44. Members of the lconv Struct

Member	Description
`char *currency_symbol`	Currency symbol applicable to the locale
`char *decimal_point`	Used to format nonmonetary quantities
`char *grouping`	Used to indicate the number of digits in each nonmonetary grouping
`char *int_curr_symbol`	Used as international currency symbol (ISO 4217:1987) for that particular locale plus the symbol used to separate the currency symbol from the monetary quantity

Table 1-44. Members of the lconv Struct (Cont'd)

Member	Description
`char *mon_decimal_point`	Used for decimal point format monetary quantities
`char *mon_grouping`	Used to indicate the number of digits in each monetary grouping
`char *mon_thousands_sep`	Used to group monetary quantities prior to the decimal point
`char *negative_sign`	Used to indicate a negative monetary quantity
`char *positive_sign`	Used to indicate a positive monetary quantity
`char *thousands_sep`	Used to group nonmonetary quantities prior to the decimal point
`char frac_digits`	Number of digits displayed after the decimal point in monetary quantities in other than international format
`char int_frac_digits`	Number of digits displayed after the decimal point in international monetary quantities
`char p_cs_precedes`	If set to 1, the `currency_symbol` precedes the positive monetary quantity. If set to 0, the `currency_symbol` succeeds the positive monetary quantity.
`char n_cs_precedes`	If set to 1, the `currency_symbol` precedes the negative monetary quantity. If set to 0, the `currency_symbol` succeeds the negative monetary quantity.
`char n_sign_posn`	Indicates the positioning of `negative_sign` for monetary quantities.
`char n_sep_by_space`	If set to 1, the `currency_symbol` is separated from the negative monetary quantity. If set to 0, the `currency_symbol` is not separated from the negative monetary quantity.
`char p_sep_by_space`	If set to 1, the `currency_symbol` is separated from the positive monetary quantity. If set to 0, the `currency_symbol` is not separated from the positive monetary quantity.

For `grouping` and `non_grouping`, an element of `CHAR_MAX` indicates that no further grouping will be performed, a 0 indicates that the previous element should be used to group the remaining digits, and any other integer value is used as the number of digits in the current grouping.

The definitions of the values for `p_sign_posn` and `n_sign_posn` are:

- parentheses surround `currency_symbol` and quantity
- sign string precedes `currency_symbol` and quantity
- sign string succeeds `currency_symbol` and quantity
- sign string immediately precedes `currency_symbol`
- sign string immediately succeeds `currency_symbol`

Error Conditions

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

Example

```
#include <locale.h>

struct lconv *c_locale;

c_locale = localeconv ();     /* Only the C locale is */
                              /* currently supported  */
```

See Also

setlocale

localtime

Convert calendar time into broken-down time

Synopsis

```
#include <time.h>
struct tm *localtime(const time_t *t);
```

Description

The `localtime` function converts a pointer to a calendar time into a broken-down time that corresponds to current time zone. A broken-down time is a structured variable, which is described in "time.h" on page 1-35. This implementation of the header file does not support the Daylight Saving flag nor does it support time zones and, thus, `localtime` is equivalent to the `gmtime` function.

The broken-down time is returned by `localtime` as a pointer to static memory, which may be overwritten by a subsequent call to either `localtime`, or to `gmtime`.

Error Conditions

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

Example

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

time_t cal_time;
struct tm *tm_ptr;

cal_time = time(NULL);
if (cal_time != (time_t) -1) {
    tm_ptr = localtime(&cal_time);
```

```
    printf("The year is %4d\n",1900 + (tm_ptr->tm_year));
}
```

See Also

asctime, gmtime, mktime, time

log

Natural logarithm

Synopsis

```
#include <math.h>

float logf (float x);
double log (double x);
long double logd (long double x);
```

Description

The natural logarithm functions compute the natural (base e) logarithm of their argument.

Error Conditions

The natural logarithm functions return zero and set `errno` to `EDOM` if the input value is zero or negative.

Example

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

y = log (1.0);          /* y = 0.0 */
x = logf (2.71828);     /* x = 1.0 */
```

See Also

alog, exp, log10

log10

Base 10 logarithm

Synopsis

```
#include <math.h>

float log10f (float x);
double log10 (double x);
long double log10d (long double x);
```

Description

The `log10` functions produce the base 10 logarithm of their argument.

Error Conditions

The `log10` functions indicate a domain error (set `errno` to `EDOM`) and return zero if the input is zero or negative.

Example

```
#include <math.h>

double y;
float x;

y = log10 (100.0);     /* y = 2.0 */
x = log10f (10.0);     /* x = 1.0 */
```

See Also

alog, log, pow

longjmp

Second return from `setjmp`

Synopsis

```
#include <setjmp.h>
void longjmp (jmp_buf env, int return_val);
```

Description

The `longjmp` function causes the program to execute a second return from the place where `setjmp (env)` was called (with the same `jmp_buf` argument).

The `longjmp` function takes as its arguments a jump buffer that contains the context at the time of the original call to `setjmp`. It also takes an integer, `return_val`, which `setjmp` returns if `return_val` is non-zero. Otherwise, `setjmp` returns a 1.

If `env` was not initialized through a previous call to `setjmp` or the function that called `setjmp` has since returned, the behavior is undefined.

The use of `setjmp` and `longjmp` (or similar functions which do not follow conventional C/C++ flow control) may produce unexpected results when the application is compiled with optimizations enabled under certain circumstances. Functions that call `setjmp` or `longjmp` are optimized by the compiler with the assumption that all variables referenced may be modified by any functions that are called. This assumption ensures that it is safe to use `setjmp` and `longjmp` with optimizations enabled, though it does mean that it is dangerous to conceal from the optimizer that a call to `setjmp` or `longjmp` is being made, for example by calling through a function pointer.

Error Conditions

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

Example

```
#include <setjmp.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

jmp_buf env;
int res;
void func (void);

main() {
if ((res = setjmp(env)) != 0) {
    printf ("Problem %d reported by func ()\n", res);
    exit (EXIT_FAILURE);
  }
func();
}

void func (void) {
if (errno != 0) {
    longjmp (env, errno);
  }
}
```

See Also

setjmp

malloc

Allocate memory

Synopsis

```
#include <stdlib.h>
void *malloc (size_t size);
```

Description

The `malloc` function returns a pointer to a block of memory of length `size`. The block of memory is uninitialized.

The object is allocated from the current heap, which is the default heap unless `set_alloc_type` or `heap_switch` has been called to change the current heap to an alternate heap.

Error Conditions

The `malloc` function returns a null pointer if it is unable to allocate the requested memory.

Example

```
#include <stdlib.h>

int *ptr;

ptr = (int *)malloc (10);     /* ptr points to an   */
                              /* array of length 10 */
```

See Also

calloc, free, heap_calloc, heap_free, heap_lookup_name, heap_malloc, heap_realloc, realloc, set_alloc_type

max

Maximum

Synopsis

```
#include <stdlib.h>
int max (int value1, int value2);
```

Description

The `max` function is an Analog Devices extension to the ANSI standard.

The `max` function returns the larger of its two arguments. The `max` function is a built-in function which is implemented with an `Rn = MAX(Rx,Ry)` instruction.

Error Conditions

The `max` function does not return an error code.

Example

```
#include <stdlib.h>

int i;

i = max (10, 8);     /* returns 10 */
```

See Also

fmax, fmin, llmax, llmin, lmax, lmin, min

memchr

Find first occurrence of character

Synopsis

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

Description

The `memchr` function compares the range of memory pointed to by `s1` with the input character `c` and returns a pointer to the first occurrence of `c`. A null pointer is returned if `c` does not occur in the first `n` characters.

Error Conditions

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

Example

```
#include <string.h>

char *ptr;

ptr = memchr ("TESTING", 'E', 7);
    /* ptr points to the E in TESTING */
```

See Also

strchr, strrchr

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 value if the `s1` object is lexically greater than the `s2` object, a negative value if the `s2` object is lexically greater than the `s1` object, and a zero 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";
int result;

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

See Also

strcmp, strcoll, strncmp

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. For more information, see "memmove" on page 1-258.

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";

memcpy (b, a, 3);     /* b = "SRCT" */
```

See Also

memmove, strcpy, strncpy

memmove

Copy characters between overlapping objects

Synopsis

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

Description

The `memmove` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The entire object is copied correctly even if the objects overlap.

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

Error Conditions

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

Example

```
#include <string.h>

char *ptr, *str = "ABCDE";

ptr = str + 2;
memmove (ptr, str, 3);     /* ptr = "ABC", str = "ABABC" */
```

See Also

memcpy, strcpy, strncpy

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

min

Minimum

Synopsis

```
#include <stdlib.h>
int min (int value1, int value2);
```

Description

The `min` function is an Analog Devices extension to the ANSI standard.

The `min` function returns the smaller of its two arguments. The `min` function is a built-in function which is implemented with an `Rn=MIN(Rx,Ry)` instruction.

Error Conditions

The `min` function does not return an error code.

Example

```
#include <stdlib.h>

int i;

i = min (10, 8);     /* returns 8 */
```

See Also

fmin, llmax, llmin, lmax, lmin, max

mktime

Convert broken-down time into a calendar time

Synopsis

```
#include <time.h>
time_t mktime(struct tm *tm_ptr);
```

Description

The `mktime` function converts a pointer to a broken-down time, which represents a local date and time, into a calendar time. However, this implementation of `time.h` does not support either daylight saving or time zones and hence this function will interpret the argument as Coordinated Universal Time (UTC).

A broken-down time is a structured variable which is defined in the `time.h` header file as:

```
struct tm { int tm_sec;   /* seconds after the minute [0,61] */
   int tm_min;            /* minutes after the hour [0,59]   */
   int tm_hour;           /* hours after midnight [0,23]     */
   int tm_mday;           /* day of the month [1,31]         */
   int tm_mon;            /* months since January [0,11]     */
   int tm_year;           /* years since 1900                */
   int tm_wday;           /* days since Sunday [0, 6]        */
   int tm_yday;           /* days since January 1st [0,365]  */
   int tm_isdst;          /* Daylight Saving flag            */
};
```

The various components of the broken-down time are not restricted to the ranges indicated above. The `mktime` function calculates the calendar time from the specified values of the components (ignoring the initial values of `tm_wday` and `tm_yday`), and then “normalizes” the broken-down time forcing each component into its defined range.

If the component tm_isdst is zero, then the mktime function assumes that daylight saving is not in effect for the specified time. If the component is set to a positive value, then the function assumes that daylight saving is in effect for the specified time and will make the appropriate adjustment to the broken-down time. If the component is negative, the mktime function should attempt to determine whether daylight saving is in effect for the specified time but because neither time zones nor daylight saving are supported, the effect will be as if tm_isdst were set to zero.

Error Conditions

The mktime function returns the value ((time_t) -1) if the calendar time cannot be represented.

Example

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

static const char *wday[] = {"Sun","Mon","Tue","Wed",
                             "Thu","Fri","Sat","???"};

struct tm tm_time = {0,0,0,0,0,0,0,0,0};

tm_time.tm_year = 2000 - 1900;
tm_time.tm_mday = 1;

if (mktime(&tm_time) == -1)
    tm_time.tm_wday = 7;
printf("%4d started on a %s\n",
    1900 + tm_time.tm_year,
    wday[tm_time.tm_wday]);
```

See Also

gmtime, localtime, time

modf

Separate integral and fractional parts

Synopsis

```
#include <math.h>

float modff (float x, float *intptr);
double modf (double x, double *intptr);
long double modfd (long double x, long double *intptr);
```

Description

The `modf` functions separate 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 `intptr`. The integral and fractional portions have the same sign as the input.

Error Conditions

The `modf` functions do not return error conditions.

Example

```
#include <math.h>

double y, n;
float m, p;

y = modf (-12.345, &n);     /* y = -0.345, n = -12.0 */
m = modff (11.75, &p);      /* m = 0.75, p = 11.0    */
```

See Also

frexp

mulifx

Multiplication of integer by fixed-point to give integer result

Synopsis

```
#include <stdfix.h>

int mulir(int a, fract b);
long int mulilr(long int a, long fract b);
unsigned int muliur(unsigned int a, unsigned fract b);
unsigned long int muliulr(unsigned long int a,
                          unsigned long fract b);
```

Description

Given an integer and a fixed-point value, the `muli`*fx* family of functions computes the product and returns the closest integer value to the result.

Error Conditions

The `muli`*fx* family of functions do not return error conditions.

Example

```
#include <stdfix.h>
unsigned long int ulprod;
ulprod = muliulr(128, 0.125ulr);        /* ulquo == 16 */
```

See Also

No related functions.

perror

Print an error message on standard error stream

Synopsis

```
#include <stdio.h>
void perror(const char *s);
```

Description

The `perror` function is used to output an error message to the standard stream `stderr`.

If the string `s` is not a null pointer and if the first character addressed by `s` is not a null character, then the function will output the string `s` followed by the character sequence ": ". The function will then print the message that is associated with the current value of `errno`. Note that the message "`no error`" is used if the value of `errno` is zero.

Error Conditions

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

Example

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

float x;

x = acosf (1234.5); /* domain of acosf is [-1.0,1.0] */;
if (errno != 0)
      perror("acosf failure");
```

See Also

strerror

pow

Raise to a power

Synopsis

```
#include <math.h>

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

Description

The power functions compute the value of the first argument raised to the power of the second argument.

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, zero is returned.

Example

```
#include <math.h>

double z;
float x;

z = pow (5.0, 2.0);      /* z = 16.0 */
x = powf (5.0, 2.0);     /* x = 16.0 */
```

See Also

exp, ldexp

printf

Print formatted output

Synopsis

```
#include <stdio.h>
int printf(const char *format, /* args*/ ...);
```

Description

The `printf` function places output on the standard output stream `stdout` in a form specified by `format`. The `printf` function is equivalent to `fprintf` with the `stdout` passed as the first argument. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to `fprintf` (on page 1-153) for a description of the valid format specifiers.

The `printf` function returns the number of characters transmitted.

Error Conditions

If the `printf` function is unsuccessful, a negative value is returned.

Example

```
#include <stdio.h>

void printf_example(void)
{
   int arg = 255;
   /* Output will be "hex:ff, octal:377, integer:255" */
   printf("hex:%x, octal:%o, integer:%d\n", arg, arg, arg);
}
```

See Also

fprintf

putc

Put a character on a stream

Synopsis

```
#include <stdio.h>
int putc(int ch, FILE *stream);
```

Description

The `putc` function writes its argument to the output stream pointed to by `stream`, after converting `ch` from an `int` to an `unsigned char`.

If the `putc` function call is successful `putc` returns its argument `ch`.

Error Conditions

The stream's error indicator will be set if the call is unsuccessful, and the function will return `EOF`.

Example

```
#include <stdio.h>

void putc_example(void)
{
   /* write the character 'a' to stdout */
   if (putc('a', stdout) == EOF)
      fprintf(stderr, "putc failed\n");
}
```

See Also

fputc

putchar

Write a character to `stdout`

Synopsis

```
#include <stdio.h>
int putchar(int ch);
```

Description

The `putchar` function writes its argument to the standard output stream, after converting `ch` from an `int` to an `unsigned char`. A call to `putchar` is equivalent to calling `putc(ch, stdout)`.

If the `putchar` function call is successful `putchar` returns its argument `ch`.

Error Conditions

The stream's error indicator will be set if the call is unsuccessful, and the function will return `EOF`.

Example

```
#include <stdio.h>

void putchar_example(void)
{
   /* write the character 'a' to stdout */
   if (putchar('a') == EOF)
      fprintf(stderr, "putchar failed\n");
}
```

See Also

putc

puts

Put a string to `stdout`

Synopsis

```
#include <stdio.h>
int puts(const char *s);
```

Description

The `puts` function writes the string pointed to by `s`, followed by a `NEWLINE` character, to the standard output stream `stdout`. The terminating null character of the string is not written to the stream.

If the function call is successful then the return value is zero or greater.

Error Conditions

The macro `EOF` is returned if `puts` was unsuccessful, and the error indicator for `stdout` will be set.

Example

```
#include <stdio.h>

void puts_example(void)
{
   /* write the string "example" to stdout */
   if (puts("example") < 0)
      fprintf(stderr, "puts failed\n");
}
```

See Also

fputs

qsort

Quicksort

Synopsis

```
#include <stdlib.h>

void qsort (void *base, size_t nelem, size_t size,
            int (*compar) (const void *, const void *));
```

Description

The `qsort` function sorts an array of `nelem` objects, pointed to by `base`. The size of each object is specified by `size`.

The contents of the array are sorted into ascending order according to a comparison function pointed to by `compar`, which is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two elements compare as equal, their order in the sorted array is unspecified. The `qsort` function executes a binary-search operation on a pre-sorted array, where

- `base` points to the start of the array.
- `nelem` is the number of elements in the array.
- `size` is the size of each element of the array.
- `compar` is a pointer to a function that is called by `qsort` to compare two elements of the array. The function should return a value less than, equal to, or greater than zero, according to whether the first argument is less than, equal to, or greater than the second.

Error Conditions

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

Example

```
#include <stdlib.h>

float a[10];

int compare_float (const void *a, const void *b)
{
    float aval = *(float *)a;
    float bval = *(float *)b;
    if (aval < bval)
        return -1;
    else if (aval == bval)
        return 0;
    else
        return 1;
}

qsort (a, sizeof (a)/sizeof (a[0]), sizeof (a[0]),
compare_float);
```

See Also

bsearch

raise

Force a signal

Synopsis

```
#include <signal.h>

int raise (int sig);
int raisensm(int sig);
```

Description

The `raise` function is an Analog Devices extension to the ANSI standard.

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 signals listed in Table 1-30 on page 1-108, Table 1-31 on page 1-109, Table 1-32 on page 1-110, Table 1-33 on page 1-112, and Table 1-35 on page 1-115.

ⓘ The `raise` function uses self-modifying code. If this is not suitable for your application, then use the `raisensm` function instead. The choice of function has no effect on the dispatcher used and no effect on the overall interrupt handling performance.

Error Conditions

The `raise` function returns a zero if successful or a non-zero value if it fails.

Example

```
#include <signal.h>

raise (SIG_IRQ2);    /* invoke the interrupt 2 handler */
```

See Also

interrupt, signal

rand

Random number generator

Synopsis

```
#include <stdlib.h>
int rand (void);
```

Description

The `rand` function returns a pseudo-random integer value in the range $[0, 2^{31} - 1]$.

For this function, the measure of randomness is its periodicity, the number of values it is likely to generate before repeating a pattern. The output of the pseudo-random number generator has a period in the order of $2^{31} - 1$.

Error Conditions

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

Example

```
#include <stdlib.h>

int i;
i = rand ();
```

See Also

srand

read_extmem

Read external memory

Synopsis

```
#include <21261.h>
#include <21262.h>
#include <21266.h>
#include <21267.h>
#include <21362.h>
#include <21363.h>
#include <21364.h>
#include <21365.h>
#include <21366.h>

void read_extmem(void    *internal_address,
                 void    *external_address,
                 size_t  n);
```

Description

On ADSP-2126x and some ADSP-2136x processors, it is not possible for the core to access external memory directly. The `read_extmem` function copies data from external to internal memory.

The `read_extmem` function will transfer `n` 32-bit words from `external_address` to `internal_address`.

Error Conditions

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

Example

```
#include <21262.h>

int intmem1[100];
int intmem2[100];

/* Place extmem1 in external memory, in the user-defined */
/* section "seg_extmem"                                  */
#pragma section("seg_extmem", DMA_ONLY)
int extmem1[100];

/* Place extmem2 in external memory, in the user-defined */
/* section "seg_extmem"                                  */
#pragma section("seg_extmem", DMA_ONLY)
int extmem2[100];

main() {
/* Transfer 100 words from external memory to internal memory */
   read_extmem(intmem1, extmem1, 100);

/* Transfer 100 words from external memory to internal memory */
   write_extmem(intmem2, extmem2, 100);
}
```

This example requires a customized `.ldf` file containing a section, `seg_extmem`, that resides in external memory.

See Also

write_extmem

realloc

Change memory allocation

Synopsis

```
#include <stdlib.h>
void *realloc (void *ptr, size_t size);
```

Description

The `realloc` function changes the memory allocation of the object pointed to by `ptr` to `size`. Initial values for the new object are taken from those in the object pointed to by `ptr`:

- If the size of the new object is greater than the size of the object pointed to by `ptr`, then the values in the newly allocated section are undefined.
- If `ptr` is a non-null pointer that was not allocated with `malloc` or `calloc`, the behavior is undefined.
- If `ptr` is a null pointer, `realloc` imitates `malloc`. If `size` is zero and `ptr` is not a null pointer, `realloc` imitates `free`.
- If `ptr` is not a null pointer, then the object is reallocated from the heap that the object was originally allocated from.
- If `ptr` is a null pointer, then the object is allocated from the current heap, which is the default heap unless `set_alloc_type` or `heap_switch` has been called to change the current heap to an alternate heap.

Error Conditions

If memory cannot be allocated, `ptr` remains unchanged and `realloc` returns a null pointer.

Example

```
#include <stdlib.h>

int *ptr;

ptr = (int *)malloc (10);       /* allocate array of 10 words */
ptr = (int *)realloc (ptr, 20); /* change size to 20 words */
```

See Also

calloc, free, heap_calloc, heap_free, heap_lookup_name, heap_malloc, heap_realloc, malloc, set_alloc_type

remove

Remove file

Synopsis

```
#include <stdio.h>
int remove(const char *filename);
```

Description

The `remove` function removes the file whose name is `filename`. After the function call, `filename` will no longer be accessible.

The `remove` function is only supported under the default device driver supplied by the VisualDSP++ simulator and EZ-KIT Lite system and it only operates on the host file system.

The `remove` function returns zero on successful completion.

Error Conditions

If the `remove` function is unsuccessful, a non-zero value is returned.

Example

```
#include <stdio.h>

void remove_example(char *filename)
{
   if (remove(filename))
      printf("Remove of %s failed\n", filename);
   else
      printf("File %s removed\n", filename);
}
```

See Also

rename

rename

Rename a file

Synopsis

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

Description

The rename function will establish a new name, using the string newname, for a file currently known by the string oldname. After a successful rename, the file will no longer be accessible by oldname.

The rename function is only supported under the default device driver supplied by the VisualDSP++ simulator and EZ-KIT Lite system and it only operates on the host file system.

If rename is successful, a value of zero is returned.

Error Conditions

If rename fails, the file named oldname is unaffected and a non-zero value is returned.

Example

```
#include <stdio.h>

void rename_file(char *new, char *old)
{
   if (rename(old, new))
      printf("rename failed for %s\n", old);
   else
      printf("%s now named %s\n", old, new);
}
```

See Also

remove

rewind

Reset file position indicator in a stream

Synopsis

```
#include <stdio.h>
void rewind(FILE *stream);
```

Description

The `rewind` function sets the file position indicator for `stream` to the beginning of the file. This is equivalent to using the `fseek` routine in the following manner:

```
fseek(stream, 0, SEEK_SET);
```

with the exception that `rewind` will also clear the error indicator.

Error Conditions

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

Example

```
#include <stdio.h>

char buffer[20];
void rewind_example(FILE *fp)
{
   /* write "a string" to a file */
   fputs("a string", fp);
   /* rewind the file to the beginning */
   rewind(fp);
   /* read back from the file - buffer will be "a string" */
   fgets(buffer, sizeof(buffer), fp);
}
```

See Also

fseek

roundfx

Round a fixed-point value to a specified precision

Synopsis

```
#include <stdfix.h>

short fract roundhr(short fract f, int n);
fract roundr(fract f, int n);
long fract roundlr(long fract f, int n);

unsigned short fract rounduhr(unsigned short fract f, int n);
unsigned fract roundur(unsigned fract f, int n);
unsigned long fract roundulr(unsigned long fract f, int n);
```

Description

The `round`*`fx`* family of functions round a fixed-point value to the number of fractional bits specified by the second argument. The rounding is round-to-nearest. If the rounded result is out of range of the result type, the result saturated to the maximum or minimum fixed-point value.

In addition to the individually-named functions for each fixed-point type, a type-generic macro `roundfx` is defined for use in C99 mode. This may be used with any of the fixed-point types and returns a result of the same type as its operand.

Error Conditions

The `round`*`fx`* family of functions do not return an error condition.

Example

```
#include <stdfix.h>
long fract f;
f = roundulr(0x12345678p-32ulr, 16);   /* f == 0x12340000ulr */
#if defined(_C99)
f = roundfx(0x12345678p-32ulr, 16);    /* f == 0x12340000ulr */
#endif
```

See Also

No related functions.

scanf

Convert formatted input from `stdin`

Synopsis

```
#include <stdio.h>
int scanf(const char *format,   /* args */...);
```

Description

The `scanf` function reads from the standard input stream `stdin`, interprets the inputs according to `format` and stores the results of the conversions in it's arguments. The string pointed to by `format` contains the control format for the input with the arguments that follow being pointers to the locations where the converted results are to be written to.

The `scanf` function is equivalent to calling `fscanf` with `stdin` as it's first argument. For details on the control format string refer to "fscanf" on page 1-168.

The `scanf` function returns number of successful conversions performed.

Error Conditions

The `scanf` function will return `EOF` if it encounters an error before any conversions are performed.

Example

```
#include <stdio.h>

void scanf_example(void)
{
   short int day, month, year;
   char string[20];
```

```
    /* Scan a string from standard input */
    scanf ("%s", string);
    /* Scan a date with any separator, eg, 1-1-2006 or 1/1/2006 */
    scanf ("%hd%*c%hd%*c%hd", &day, &month, &year);
}
```

See Also

fscanf

setbuf

Specify full buffering for a stream

Synopsis

```
#include <stdio.h>
void setbuf(FILE *stream, char* buf);
```

Description

The `setbuf` function results in the array pointed to by `buf` being used to buffer the stream pointed to by `stream` instead of an automatically allocated buffer. The `setbuf` function may be used only after the stream pointed to by `stream` is opened but before it is read or written to. Note that the buffer provided must be of size `BUFSIZ` as defined in the `stdio.h` header.

> When the buffer contains data for a text stream (either input data or output data), the information is held in the form of 8-bit characters that are packed into 32-bit memory locations. Due to internal mechanisms used to unpack and pack this data, the I/O buffer must not reside at a memory location greater than the address `0x3fffffff`.

If `buf` is the `NULL` pointer, the input/output will be completely unbuffered.

Error Conditions

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

Example

```
#include <stdio.h>

#include <stdlib.h>
void* allocate_buffer_from_heap(FILE* fp)
```

```
{
   /* Allocate a buffer from the heap for the file pointer */
   void* buf = malloc(BUFSIZ);
   if (buf != NULL)
      setbuf(fp, buf);
   return buf;
}
```

See Also

setvbuf

setjmp

Define a run-time label

Synopsis

```
#include <setjmp.h>
int setjmp (jmp_buf env);
```

Description

The `setjmp` function saves the calling environment in the `jmp_buf` argument. The effect of the call is to declare a run-time label that can be jumped to via a subsequent call to `longjmp`.

When `setjmp` is called, it immediately returns with a result of zero to indicate that the environment has been saved in the `jmp_buf` argument. If, at some later point, `longjmp` is called with the same `jmp_buf` argument, `longjmp` restores the environment from the argument. The execution is then resumed at the statement immediately following the corresponding call to `setjmp`. The effect is as if the call to `setjmp` has returned for a second time but this time the function returns a non-zero result.

The effect of calling `longjmp` is undefined if the function that called `setjmp` has returned in the interim.

The use of `setjmp` and `longjmp` (or similar functions which do not follow conventional C/C++ flow control) may produce unexpected results when the application is compiled with optimizations enabled under certain circumstances. Functions that call `setjmp` or `longjmp` are optimized by the compiler with the assumption that all variables referenced may be modified by any functions that are called. This assumption ensures that it is safe to use `setjmp` and `longjmp` with optimizations enabled, though it does mean that it is dangerous to conceal from the optimizer that a call to `setjmp` or `longjmp` is being made, for example by calling through a function pointer.

Error Conditions

The label `setjmp` does not return an error condition.

Example

See "longjmp" on page 1-251

See Also

longjmp

setlocale

Set the current locale

Synopsis

```
#include <locale.h>
char *setlocale (int category, const char *locale);
```

Description

The `setlocale` function uses the parameters `category` and `locale` to select a current locale. The possible values for the `category` argument are those macros defined in `locale.h` beginning with "`LC_`". The only `locale` argument supported at this time is the "`C`" locale. If a null pointer is used for the `locale` argument, `setlocale` returns a pointer to a string which is the current locale for the given category argument. A subsequent call to `setlocale` with the same `category` argument and the string supplied by the previous `setlocale` call returns the locale to its original status. The string pointed to may not be altered by the program but may be overwritten by subsequent `setlocale` calls.

Error Conditions

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

Example

```
#include <locale.h>

setlocale (LC_ALL, "C");
    /* sets the locale to the "C" locale */
```

See Also

localeconv

setvbuf

Specify buffering for a stream

Synopsis

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

Description

The `setvbuf` function may be used after a stream has been opened but before it is read or written to. The kind of buffering that is to be used is specified by the `type` argument. The valid values for `type` are detailed in Table 1-45.

Table 1-45. Valid Values for type

Type	Effect
`_IOFBF`	Use full buffering for output. Only output to the host system when the buffer is full, or when the stream is flushed or closed, or when a file positioning operation intervenes.
`_IOLBF`	Use line buffering. The buffer will be flushed whenever a `NEWLINE` is written, as well as when the buffer is full, or when input is requested.
`_IONBF`	Do not use any buffering at all.

If `buf` is not the `NULL` pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. Note that if `buf` is non-`NULL` then you must ensure that the associated storage continues to be available until you close the stream identified by `stream`. The `size` argument specifies the size of the buffer required. If input/output is unbuffered, the `buf` and `size` arguments are ignored.

> When the buffer contains data for a text stream (either input data or output data), the information is held in the form of 8-bit characters that are packed into 32-bit memory locations. Due to internal mechanisms used to unpack and pack this data, the I/O buffer must not reside at a memory location greater than the address `0x3fffffff`.

If `buf` is the `NULL` pointer, buffering is enabled and a buffer of size `size` will be automatically generated.

The `setvbuf` function returns zero when successful.

Error Conditions

The `setvbuf` function will return a non-zero value if either an invalid value is given for `type`, or if the stream has already been used to read or write data, or if an I/O buffer could not be allocated.

Example

```
#include <stdio.h>

void line_buffer_stderr(void)
{
   /* stderr is not buffered - set to use line buffering */
   setvbuf (stderr,NULL,_IOLBF,BUFSIZ);
}
```

See Also

setbuf

set_alloc_type

Set heap for dynamic memory allocation

Synopsis

```
#include <stdlib.h>
int set_alloc_type(char * heap_name);
```

Description

The `set_alloc_type` function is an Analog Devices extension to the ANSI standard.

The `set_alloc_type` function specifies a heap from which `malloc` and `calloc` should subsequently allocate memory. The `heap_name` argument should be the name of the segment containing the heap as a string. For more information on creating multiple heaps, see Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, section "Using Multiple Heaps".

The `set_alloc_type` function is not available in multithreaded environments.

Error Conditions

The `set_alloc_type` function returns a non-zero value if the heap specified cannot be found.

Example

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

char *mymem, *stdmem;

int allocate()
{
```

```
  int res;
  res = set_alloc_type("seg_heaq");
  if (res != 0) {
     printf("Failed to switch heaps\n");
     return 1;
  }

  mymem = malloc(10);     /* mymem is allocated on "seg_heaq" */
  if (mymem == NULL) {
      printf("Failed to allocate memory from seg_heaq\n");
      return 1;
  }

  res = set_alloc_type("seg_heap");
  if (res != 0) {
      printf("Failed to switch heaps\n");
      return 1;
  }

  stdmem = malloc(10); /* stdmem is allocated on default heap */
  if (stdmem == NULL) {
      printf("Failed to allocate memory from the default
heap\n");
      return 1;
  }

  printf("Memory was allocated at %p %p\n", mymem, stdmem);
  return 0;
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_lookup_name, heap_malloc, heap_realloc, malloc, realloc

signal

Define signal handling

Synopsis

```
#include <signal.h>

void (*signal (int sig, void (*func)(int))) (int);
void (*signalnsm (int sig, void (*func)(int))) (int);
void (*signalf (int sig, void (*func)(int))) (int);
void (*signalfnsm (int sig, void (*func)(int))) (int);
void (*signals (int sig, void (*func)(int))) (int);
void (*signalsnsm (int sig, void (*func)(int))) (int);
void (*signalcb (int sig, void (*func)(int))) (int);
void (*signalcbnsm (int sig, void (*func)(int))) (int);
void (*signalss  (int sig, void (*func)(int))) (int);
void (*signalssnsm (int sig, void (*func)(int))) (int);
```

Description

The `signal`, `signalnsm`, `signalf`, `signalfnsm`, `signals`, `signalsnsm`, `signalcb`, `signalcbnsm`, `signalss` or `signalssnsm` functions determine how a signal that is received during program execution is handled. The specified `signal` function causes the corresponding `interrupt` dispatcher to be used when handling the interrupt (refer to “signal.h” on page 1-26 for more information).

The `signal` function returns the value of the previously installed interrupt or signal handler action. The `sig` argument must be one of the values that are listed in either Table 1-30 on page 1-108, Table 1-31 on page 1-109, Table 1-32 on page 1-110, Table 1-33 on page 1-112, or Table 1-35 on page 1-115. The `signal` function causes the receipt of the signal number `sig` to be handled in one of the ways listed in Table 1-43 on page 1-209. The function pointed to by `func` is executed once when the signal is received. Handling is then returned to the default state.

The differences between the actions taken by the supplied standard interrupt dispatchers, `interrupt`, `interruptnsm`, `interruptf`, `interruptfnsm`, `interrupts`, `interruptsnsm`, `interruptcb`, and `interruptcbnsm`, are discussed under "signal.h" on page 1-26.

Error Conditions

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

Example

```
#include <signal.h>

signal (SIG_IRQ2, irq2_handler);    /* enable interrupt 2  */
signal (SIG_IRQ2, SIG_IGN);         /* disable interrupt 2 */
```

See Also

interrupt, raise

sin

Sine

Synopsis

```
#include <math.h>

float sinf (float x);
double sin (double x);
long double sind (long double x);
```

Description

The `sin` functions return the sine of `x`. The input is interpreted as radians; the output is in the range [-1, 1].

Error Conditions

The input argument `x` for `sinf` must be in the domain [-1.647e6, 1.647e6] and the input argument for `sind` must be in the domain [-8.433e8, 8.433e8]. The functions return zero if `x` is outside their domain.

Example

```
#include <math.h>

double y;
float x;

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

See Also

asin, cos

sinh

Hyperbolic sine

Synopsis

```
#include <math.h>

float sinhf (float x);
double sinh (double x);
long double sinhd (long double x);
```

Description

The hyperbolic sine functions return the hyperbolic sine of `x`.

Error Conditions

The input argument `x` must be in the domain [-89.39, 89.39] for `sinhf`, and in the domain [-710.44, 710.44] for `sinhd`. If the input value is greater than the function's domain, then `HUGE_VAL` is returned, and if the input value is less than the domain, then `-HUGE_VAL` is returned.

Example

```
#include <math.h>

float  x;
double y;

x = sinhf ( 1.0);    /* x =  1.1752 */
y = sinh  (-1.0);    /* y = -1.1752 */
```

See Also

cosh

snprintf

Format data into an n-character array

Synopsis

```
#include <stdio.h>
int snprintf (char *str, size_t n, const char *format, ...);
```

Description

The `snprintf` function is a function that is defined in the C99 Standard (ISO/IEC 9899).

It is similar to the `sprintf` function in that `snprintf` formats data according to the argument `format`, and then writes the output to the array `str`. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to `fprintf` (on page 1-153) for a description of the valid format specifiers.

The function differs from `sprintf` in that no more than `n-1` characters are written to the output array. Any data written beyond the `n-1'th` character is discarded. A terminating `NULL` character is written after the end of the last character written to the output array unless `n` is set to zero, in which case nothing will be written to the output array and the output array may be represented by the `NULL` pointer.

The `snprintf` function returns the number of characters that would have been written to the output array `str` if `n` was sufficiently large. The return value does not include the terminating null character written to the array.

The output array will contain all of the formatted text if the return value is not negative and is also less than `n`.

Error Conditions

The `snprintf` function returns a negative value if a formatting error occurred.

Example

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

extern char *make_filename(char *name, int id)
{
   char *filename_template = "%s%d.dat";
   char *filename = NULL;

   int len = 0;
   int r;                   /* return value from snprintf  */

   do {
      r = snprintf(filename,len,filename_template,name,id);
      if (r < 0)            /* formatting error?           */
         abort();
      if (r < len)          /* was complete string written? */
         return filename;   /* return with success          */
      filename = realloc(filename,(len=r+1));
   } while (filename != NULL);
   abort();
}
```

See Also

fprintf, sprintf, vsnprintf

sprintf

Format data into a character array

Synopsis

```
#include <stdio.h>
int sprintf (char *str, const char *format, /* args */...);
```

Description

The `sprintf` function formats data according to the argument `format`, and then writes the output to the array `str`. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to `fprintf` (on page 1-153) for a description of the valid format specifiers.

In all respects other than writing to an array rather than a stream the behavior of `sprintf` is similar to that of `fprintf`.

If the `sprintf` function is successful it will return the number of characters written in the array, not counting the terminating `NULL` character.

Error Conditions

The `sprintf` function returns a negative value if a formatting error occurred.

Example

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

char filename[128];

extern char *assign_filename(char *name)
{
```

```
    char *filename_template = "%s.dat";
    int r;              /* return value from sprintf */

    if ((strlen(name)+5) > sizeof(filename))
       abort();
    r = sprintf(filename, filename_template, name);
    if (r < 0)            /* sprintf failed        */
       abort();
    return filename;  /* return with success    */
}
```

See Also

fprintf, snprintf

sqrt

Square root

Synopsis

```
#include <math.h>

float sqrtf (float x);
double sqrt (double x);
long double sqrtd (long double x);
```

Description

The square root functions return the positive square root of `x`.

Error Conditions

The square root functions return zero for negative input values and set `errno` to `EDOM` to indicate a domain error.

Example

```
#include <math.h>

double y;
float x;

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

See Also

rsqrt

srand

Random number seed

Synopsis

```
#include <stdlib.h>
void srand (unsigned int seed);
```

Description

The `srand` function is used to set the seed value for the `rand` function. A particular seed value always produces the same sequence of pseudo-random numbers.

Error Conditions

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

Example

```
#include <stdlib.h>

srand (22);
```

See Also

rand

sscanf

Convert formatted input in a string

Synopsis

```
#include <stdio.h>
int sscanf(const char *s, const char *format, /* args */...);
```

Description

The `sscanf` function reads from the string `s`. The function is equivalent to `fscanf` with the exception of the string being read from a string rather than a stream. The behavior of `sscanf` when reaching the end of the string equates to `fscanf` reaching the `EOF` in a stream. For details on the control format string, refer to "fscanf" on page 1-168.

The `sscanf` function returns the number of items successfully read.

Error Conditions

If the `sscanf` function is unsuccessful, `EOF` is returned.

Example

```
#include <stdio.h>

void sscanf_example(const char *input)
{
   short int day, month, year;
   char string[20];

   /* Scan for a string from "input" */
   sscanf (input, "%s", string);
   /* Scan a date with any separator, eg, 1-1-2006 or 1/1/2006 */
   sscanf (input, "%hd%*c%hd%*c%hd", &day, &month, &year);
}
```

See Also

fscanf

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

strchr

Find first occurrence of character in string

Synopsis

```
#include <string.h>
char *strchr (const char *s1, int c);
```

Description

The `strchr` function returns a pointer to the first location in `s1`, a null-terminated string, that contains the character `c`.

Error Conditions

The `strchr` function returns a null pointer if `c` is not part of the string.

Example

```
#include <string.h>

char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strchr (ptr1, 'E');
   /* ptr2 points to the E in TESTING */
```

See Also

memchr, strrchr

strcmp

Compare strings

Synopsis

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

Description

The `strcmp` function lexicographically compares 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

strcoll

Compare strings

Synopsis

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

Description

The `strcoll` function compares the string pointed to by `s1` with the string pointed to by `s2`. The comparison is based on the locale macro, `LC_COLLATE`. Because only the C locale is defined in the ADSP-21xxx run-time environment, the `strcoll` function is identical to the `strcmp` function. The function 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 `strcoll` function does not return an error condition.

Example

```
#include <string.h>

char string1[50], string2[50];

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

See Also

strcmp, strncmp

strcpy

Copy from one string to another

Synopsis

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

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, memmove, strncpy

strcspn

Length of character segment in one string but not the other

Synopsis

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

Description

The `strcspn` function returns the array index of the first character in `s1` which is not in the set of characters pointed to by `s2`. The order of the characters in `s2` is not significant.

Error Conditions

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

Example

```
#include <string.h>

char *ptr1, *ptr2;
size_t len;

ptr1 = "Tried and Tested";
ptr2 = "aeiou";
len = strcspn (ptr1, ptr2);     /* len = 2 */
```

See Also

strlen, strspn

strerror

Get string containing error message

Synopsis

```
#include <string.h>
char *strerror (int errnum);
```

Description

The `strerror` function is called to return a pointer to an error message that corresponds to the argument `errnum`. The global variable `errno` is commonly used as the value of `errnum`, and as `errno` is generally not supported by the library, `strerror` will always return a pointer to the string "`There are no error strings defined!`".

Error Conditions

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

Example

```
#include <string.h>

char *ptr1;

ptr1 = strerror (1);
```

See Also

No related function.

strftime

Format a broken-down time

Synopsis

```
#include <time.h>

size_t strftime(char *buf,
                size_t buf_size,
                const char *format,
                const struct tm *tm_ptr);
```

Description

The `strftime` function formats the broken-down time `tm_ptr` into the `char` array pointed to by `buf`, under the control of the format string `format`. At most, `buf_size` characters (including the null terminating character) are written to `buf`.

In a similar way as for `printf`, the format string consists of ordinary characters, which are copied unchanged to the `char` array `buf`, and zero or more conversion specifiers. A conversion specifier starts with the character `%` and is followed by a character that indicates the form of transformation required – the supported transformations are given in Table 1-46.

Note that the `strftime` function only supports the "C" locale, and this is reflected in the table.

Table 1-46. Conversion Specifiers Supported by strftime

Conversion Specifier	Transformation	ISO/IEC 9899
`%a`	abbreviated weekday name	yes
`%A`	full weekday name	yes
`%b`	abbreviated month name	yes
`%B`	full month name	yes

Table 1-46. Conversion Specifiers Supported by strftime (Cont'd)

Conversion Specifier	Transformation	ISO/IEC 9899
%c	date and time presentation in the form of DDD MMM dd hh:mm:ss yyyy	yes
%C	century of the year	POSIX.2-1992 + ISO C99
%d	day of the month (01 - 31)	yes
%D	date represented as mm/dd/yy	POSIX.2-1992 + ISO C99
%e	day of the month, padded with a space character (cf %d)	POSIX.2-1992 + ISO C99
%F	date represented as yyyy-mm-dd	POSIX.2-1992 + ISO C99
%h	abbreviated name of the month (same as %b)	POSIX.2-1992 + ISO C99
%H	hour of the day as a 24-hour clock (00-23)	yes
%I	hour of the day as a 12-hour clock (00-12)	yes
%j	day of the year (001-366)	yes
%k	hour of the day as a 24-hour clock padded with a space (0-23)	no
%l	hour of the day as a 12-hour clock padded with a space (0-12)	no
%m	month of the year (01-12)	yes
%M	minute of the hour (00-59)	yes
%n	newline character	POSIX.2-1992 + ISO C99
%p	AM or PM	yes
%P	am or pm	no
%r	time presented as either hh:mm:ss AM or as hh:mm:ss PM	POSIX.2-1992 + ISO C99
%R	time presented as hh:mm	POSIX.2-1992 + ISO C99
%S	second of the minute (00-61)	yes
%t	tab character	POSIX.2-1992 + ISO C99

Table 1-46. Conversion Specifiers Supported by strftime (Cont'd)

Conversion Specifier	Transformation	ISO/IEC 9899
`%T`	time formatted as `%H:%M:%S`	POSIX.2-1992 + ISO C99
`%U`	week number of the year (week starts on Sunday) (`00-53`)	yes
`%w`	weekday as a decimal (`0-6`) (`0` if Sunday)	yes
`%W`	week number of the year (week starts on Sunday) (`00-53`)	yes
`%x`	date represented as `mm/dd/yy` (same as `%D`)	yes
`%X`	time represented as `hh:mm:ss`	yes
`%y`	year without the century (`00-99`)	yes
`%Y`	year with the century (`nnnn`)	yes
`%Z`	the time zone name, or nothing if the name cannot be determined	yes
`%%`	`%` character	yes

The current implementation of `time.h` does not support time zones and, therefore, the `%Z` specifier does not generate any characters.

The `strftime` function returns the number of characters (not including the terminating null character) that have been written to `buf`.

Error Conditions

The `strftime` function returns zero if more than `buf_size` characters are required to process the format string. In this case, the contents of the array `buf` will be indeterminate.

Example

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

extern void
print_time(time_t tod)
{
   char tod_string[100];

   strftime(tod_string,
            100,
            "It is %M min and %S secs after %l o'clock (%p)",
            gmtime(&tod));
   puts(tod_string);
}
```

See Also

ctime, gmtime, localtime, mktime

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 `s1` (not including the terminating null character).

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

strcspn, strspn

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 character (`'\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

strncmp

Compare characters in strings

Synopsis

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

Description

The `strncmp` function lexicographically performs the comparison on the first `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

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, starting with element 0, pointed to by s2 into the space pointed to by s1. If the last character copied from s2 is not a null, the result does 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 the null character 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, memmove, strcpy

strpbrk

Find character match in two strings

Synopsis

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

Description

The `strpbrk` function returns a pointer to the first character in `s1` that is also found in `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

Error Conditions

In the event that no character in `s1` matches any in `s2`, a null pointer is returned.

Example

```
#include <string.h>

char *ptr1, *ptr2, *ptr3;

ptr1 = "TESTING";
ptr2 = "SHOP"
ptr3 = strpbrk (ptr1, ptr2);
   /* ptr3 points to the S in TESTING */
```

See Also

strspn

strrchr

Find last occurrence of character in string

Synopsis

```
#include <string.h>
char *strrchr (const char *s1, int c);
```

Description

The `strrchr` function returns a pointer to the last occurrence of character `c` in the null-terminated input string `s1`.

Error Conditions

The `strrchr` function returns a null pointer if `c` is not found.

Example

```
#include <string.h>

char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strrchr (ptr1, 'T');
   /* ptr2 points to the second T of TESTING */
```

See Also

memchr, strchr

strspn

Length of segment of characters in both strings

Synopsis

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

Description

The `strspn` function returns the array index of the first character in `s1` which is in the set of characters pointed to by `s2`. The order of the characters in `s2` is not significant.

Error Conditions

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

Example

```
#include <string.h>

size_t len;
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = "ERST";
len = strspn (ptr1, ptr2);    /* len = 4 */
```

See Also

strcspn, strlen

strstr

Find string within string

Synopsis

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

Description

The `strstr` function returns a pointer to the first occurrence in the string pointed to by `s1` of the characters in the string pointed to by `s2`. This excludes the terminating null character in `s1`.

Error Conditions

If the string is not found, `strstr` returns a null pointer. If `s2` points to a string of zero length, `s1` is returned.

Example

```
#include <string.h>

char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strstr (ptr1, "E");
   /* ptr2 points to the E in TESTING */
```

See Also

strchr

strtod

Convert string to double

Synopsis

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr)
```

Description

The `strtod` function extracts a value from the string pointed to by `nptr`, and returns the value as a `double`. The `strtod` function expects `nptr` to point to a string that represents either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (+) or minus (-); and digits are one or more decimal digits. The sequence of digits may contain a decimal point (.).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign]
[digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (-) followed by the hexadecimal prefix `0x` or `0X`. This character

sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P`, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number stops the scan. If `endptr` is not NULL, a pointer to the character that stopped the scan is stored at the location pointed to by `endptr`. If no conversion can be performed, the value of `nptr` is stored at the location pointed to by `endptr`.

Error Conditions

The `strtod` function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`. If the correct value results in an overflow, a positive or negative (as appropriate) `HUGE_VAL` is returned. If the correct value results in an underflow, zero is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

Example

```
#include <stdlib.h>

char *rem;
double dd;

dd = strtod ("2345.5E4 abc",&rem);
     /* dd = 2.3455E+7, rem = " abc" */


dd = strtod ("-0x1.800p+9,123",&rem);
     /* dd = -768.0, rem = ",123"   */
```

See Also

atof, strtofxfx, strtol, strtoul

strtofxfx

Convert string to fixed-point

Synopsis

```
#include <stdfix.h>

short fract strtofxhr(const char *nptr, char **endptr);
fract strtofxr(const char *nptr, char **endptr);
long fract strtofxlr(const char *nptr, char **endptr);

unsigned short fract strtofxuhr(const char *nptr, char **endptr);
unsigned fract strtofxur(const char *nptr, char **endptr);
unsigned long fract strtofxulr(const char *nptr, char **endptr);
```

Description

The `strtofx`*`fx`* family of functions extracts a value from the string pointed to by `nptr`, and converts the value to a fixed-point representation. The `strtofx`*`fx`* functions expect `nptr` to point to a string that represents either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (+) or minus (–); and digits are one or more decimal digits. The sequence of digits may contain a decimal point (.).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

`[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]`

A hexadecimal floating-point number may start with an optional plus (+) or minus (–) followed by the hexadecimal prefix `0x` or `0X`. This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P`, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number stops the scan. If `endptr` is not NULL, a pointer to the character that stopped the scan is stored at the location pointed to by `endptr`. If no conversion can be performed, the value of `nptr` is stored at the location pointed to by `endptr`.

Error Conditions

The `strtofx`*`fx`* functions return a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`. If the correct value results in an overflow, the maximum positive or negative (as appropriate) fixed-point value is returned. If the correct value results in an underflow, zero is returned. The `ERANGE` value is stored in `errno` in the case of overflow.

Example

```
#include <stdfix.h>
char *rem;
unsigned long fract ulr;

ulr = strtofxulr ("0x180p-12,123",&rem);
     /* ulr = 0x1800p-16ulr, rem = ",123"   */
```

See Also

strtod, strtol, strtoul, strtoull

strtok

Convert string to tokens

Synopsis

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

Description

The `strtok` function returns successive tokens from the string `s1`, where each token is delimited by characters from `s2`.

A call to `strtok` with `s1` not NULL returns a pointer to the first token in `s1`, where a token is a consecutive sequence of characters not in `s2`. `s1` is modified in place to insert a null character at the end of the token returned. If `s1` consists entirely of characters from `s2`, NULL is returned.

Subsequent calls to `strtok` with `s1` equal to NULL return successive tokens from the same string. When the string contains no further tokens, NULL is returned. Each new call to `strtok` may use a new delimiter string, even if `s1` is NULL. If `s1` is NULL, the remainder of the string is converted into tokens using the new delimiter characters.

Error Conditions

The `strtok` function returns a null pointer if there are no tokens remaining in the string.

Example

```
#include <string.h>

static char str[] = "a phrase to be tested, today";
char *t;
```

```
t = strtok (str, " ");      /* t points to "a"            */
t = strtok (NULL, " ");     /* t points to "phrase"       */
t = strtok (NULL, ",");     /* t points to "to be tested" */
t = strtok (NULL, ".");     /* t points to " today"       */
t = strtok (NULL, ".");     /* t = NULL                   */
```

See Also

No related function.

strtol

Convert string to long integer

Synopsis

```
#include <stdlib.h>
long int strtol (const char *nptr, char **endptr, int base);
```

Description

The `strtol` function returns as a `long int` the value represented by the string `nptr`. If `endptr` is not a null pointer, `strtol` stores a pointer to the unconverted remainder in `*endptr`.

The `strtol` function breaks down the input into three sections:

- White space (as determined by `isspace`)
- Initial characters
- Unrecognized characters including a terminating null character

The initial characters may be composed of an optional sign character, `0x` or `0X` if `base` is 16, and those letters and digits which represent an integer with a radix of `base`. The letters (`a-z` or `A-Z`) are assigned the values 10 to 35, and their use is permitted only when those values are less than the value of `base`.

If `base` is zero, then the base is taken from the initial characters. A leading 0x indicates base 16; a leading 0 indicates base 8. For any other leading characters, base 10 is used. If `base` is between 2 and 36, it is used as a base for conversion.

Error Conditions

The strtol function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by endptr, provided that endptr is not a null pointer. If the correct value results in an overflow, positive or negative (as appropriate) LONG_MAX is returned. If the correct value results in an underflow, LONG_MIN is returned. ERANGE is stored in errno in the case of either overflow or underflow.

Example

```
#include <stdlib.h>

#define base 10
char *rem;
long int i;

i = strtol ("2345.5", &rem, base);
    /* i=2345, rem=".5" */
```

See Also

atoi, atol, strtofxfx. strtoll, strtoul, strtoull

strtold

Convert string to long double

Synopsis

```
#include <stdlib.h>
long double strtold(const char *nptr, char **endptr)
```

Description

The `strtold` function extracts a value from the string pointed to by `nptr`, and returns the value as a `long double`. The `strtold` function expects `nptr` to point to a string that represents either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (+) or minus (-); and digits are one or more decimal digits. The sequence of digits may contain a decimal point (.).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (-) followed by the hexadecimal prefix `0x` or `0X`. This character

sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P`, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number stops the scan. If `endptr` is not NULL, a pointer to the character that stopped the scan is stored at the location pointed to by `endptr`. If no conversion can be performed, the value of `nptr` is stored at the location pointed to by `endptr`.

Error Conditions

The `strtold` function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`. If the correct value results in an overflow, a positive or negative (as appropriate) `LDBL_MAX` is returned. If the correct value results in an underflow, zero is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

Example

```
#include <stdlib.h>

char *rem;
long double dd;

dd = strtold ("2345.5E4 abc",&rem);
     /* dd = 2.3455E+7, rem = " abc" */

dd = strtold ("-0x1.800p+9,123",&rem);
     /* dd = -768.0, rem = ",123"   */
```

See Also

atoi, atol, strtod, strtofxfx, strtoul

strtoll

Convert string to long long integer

Synopsis

```
#include <stdlib.h>
long long strtoll (const char *nptr, char **endptr, int base);
```

Description

The `strtoll` function returns as a `long long` the value represented by the string `nptr`. If `endptr` is not a null pointer, `strtoll` stores a pointer to the unconverted remainder in `*endptr`.

The `strtoll` function breaks down the input into three sections:

- White space (as determined by `isspace`)
- Initial characters
- Unrecognized characters including a terminating null character

The initial characters may be composed of an optional sign character, `0x` or `0X` if `base` is 16, and those letters and digits which represent an integer with a radix of `base`. The letters (`a-z` or `A-Z`) are assigned the values 10 to 35, and their use is permitted only when those values are less than the value of `base`.

If `base` is zero, then the base is taken from the initial characters. A leading 0x indicates base 16; a leading 0 indicates base 8. For any other leading characters, base 10 is used. If `base` is between 2 and 36, it is used as a base for conversion.

Error Conditions

The `strtoll` function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`, provided that endptr is not a null pointer. If the correct value results in an overflow, positive or negative (as appropriate) LLONG_MAX is returned. If the correct value results in an underflow, LLONG_MIN is returned. ERANGE is stored in errno in the case of either overflow or underflow.

Example

```
#include <stdlib.h>

#define base 10
char *rem;
long long i;

i = strtoll ("2345.5", &rem, base);
    /* i=2345, rem=".5" */
```

See Also

atoi, atol, strtol, strtoul, strtoull

strtoul

Convert string to unsigned long integer

Synopsis

```
#include <stdlib.h>
unsigned long int strtoul (const char *nptr, char **endptr, int
base);
```

Description

The strtoul function returns as an unsigned long int the value represented by the string nptr. If endptr is not a null pointer, strtoul stores a pointer to the unconverted remainder in *endptr.

The strtoul function breaks down the input into three sections:

- White space (as determined by isspace)
- Initial characters
- Unrecognized characters including a terminating null character

The initial characters may comprise an optional sign character, 0x or 0X, when base is 16, and those letters and digits which represent an integer with a radix of base. The letters (a-z or A-Z) are assigned the values 10 to 35, and are permitted only when those values are less than the value of base.

If base is zero, then the base is taken from the initial characters. A leading 0x indicates base 16; a leading 0 indicates base 8. For any other leading characters, base 10 is used. If base is between 2 and 36, it is used as a base for conversion.

Error Conditions

The `strtoul` function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer. If the correct value results in an overflow, `ULONG_MAX` is returned. `ERANGE` is stored in `errno` in the case of overflow.

Example

```
#include <stdlib.h>

#define base 10

char *rem;
unsigned long int i;

i = strtoul ("2345.5", &rem, base);
    /* i = 2345, rem = ".5" */
```

See Also

atoi, atol, strtofxfx, strtol, strtoll, strtoull

strtoull

Convert string to unsigned long long integer

Synopsis

```
#include <stdlib.h>
unsigned long long strtoull (const char  *nptr,
                             char       **endptr,
                             int          base);
```

Description

The `strtoull` function returns as an `unsigned long long` the value represented by the string `nptr`. If `endptr` is not a null pointer, `strtoull` stores a pointer to the unconverted remainder in `*endptr`.

The `strtoull` function breaks down the input into three sections:

- White space (as determined by `isspace`)
- Initial characters
- Unrecognized characters including a terminating null character

The initial characters may comprise an optional sign character, `0x` or `0X`, when `base` is 16, and those letters and digits which represent an integer with a radix of `base`. The letters (`a-z` or `A-Z`) are assigned the values 10 to 35, and are permitted only when those values are less than the value of `base`.

If `base` is zero, then the base is taken from the initial characters. A leading 0x indicates base 16; a leading 0 indicates base 8. For any other leading characters, base 10 is used. If `base` is between 2 and 36, it is used as a base for conversion.

Error Conditions

The `strtoull` function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer. If the correct value results in an overflow, `ULLONG_MAX` is returned. `ERANGE` is stored in `errno` in the case of overflow.

Example

```
#include <stdlib.h>

#define base 10

char *rem;
unsigned long long i;

i = strtoull ("2345.5", &rem, base);
    /* i = 2345, rem = ".5" */
```

See Also

atoi, atol, strtofxfx, strtol, strtoll, strtoul

strxfrm

Transform string using LC_COLLATE

Synopsis

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

Description

The strxfrm function transforms the string pointed to by s2 using the locale specific category LC_COLLATE. (See "setlocale" on page 1-297). It places the result in the array pointed to by s1.

> The transformation is such that if s1 and s2 were transformed and used as arguments to strcmp, the result would be identical to the result derived from strcoll using s1 and s2 as arguments. However, since only C locale is implemented, this function does not perform any transformations other than the number of characters.

The string stored in the array pointed to by s1 is never more than n characters including the terminating NULL character. strxfrm returns 1. If this returned value is n or greater, the result stored in the array pointed to by s1 is indeterminate. s1 can be a NULL pointer if n is zero.

Error Conditions

The strxfrm function does not return an error condition.

Example

```
#include <string.h>

char string1[50];

strxfrm (string1, "SOMEFUN", 49);
    /* SOMEFUN is copied into string1 */
```

See Also

setlocale, strcmp, strcoll

system

Send string to operating system

Synopsis

```
#include <stdlib.h>
int system (const char *string);
```

Description

The `system` function normally sends a string to the operating system. In the context of the ADSP-21xxx run-time environment, `system` always returns zero.

Error Conditions

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

Example

```
#include <stdlib.h>

system ("string");    /* always returns zero */
```

See Also

getenv

tan

Tangent

Synopsis

```
#include <math.h>

float tanf (float x);
double tan (double x);
long double tand (long double x);
```

Description

The tangent functions return the tangent of the argument `x`, where `x` is measured in radians.

Error Conditions

The domain of `tanf` is [-1.647e6, 1.647e6], and the domain for `tand` is [-4.21657e8 , 4.21657e8]. The functions return 0.0 if the input argument `x` is outside the respective domains.

Example

```
#include <math.h>

double y;
float x;

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

See Also

atan, atan2

tanh

Hyperbolic tangent

Synopsis

```
#include <math.h>

float tanhf (float x);
double tanh (double x);
long double tanhd (long double x);
```

Description

The hyperbolic tangent functions return the hyperbolic tangent of the argument `x`, where `x` is measured in radians.

Error Conditions

The hyperbolic tangent functions do not return an error condition.

Example

```
#include <math.h>

double x, y;
float z, w;

y = tanh (x);
z = tanhf (w);
```

See Also

cosh, sinh

time

Calendar time

Synopsis

```
#include <time.h>
time_t time(time_t *t);
```

Description

The `time` function returns the current calendar time which measures the number of seconds that have elapsed since the start of a known epoch. As the calendar time cannot be determined in this implementation of `time.h`, a result of `(time_t) -1` is returned. The function's result is also assigned to its argument, if the pointer to `t` is not a null pointer.

Error Conditions

The `time` function will return the value `((time_t) -1)` if the calendar time is not available.

Example

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

if (time(NULL) == (time_t) -1)
    printf("Calendar time is not available\n");
```

See Also

ctime, gmtime, localtime

tolower

Convert from uppercase to lowercase

Synopsis

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

Description

The `tolower` function converts the input character to lowercase if it is uppercase; otherwise, it returns the character.

Error Conditions

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

Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    if (isupper (ch))
        printf ("tolower=%#04x", tolower (ch));
    putchar ('\n');
}
```

See Also

islower, isupper, toupper

toupper

Convert from lowercase to uppercase

Synopsis

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

Description

The `toupper` function converts the input character to uppercase if it is in lowercase; otherwise, it returns the character.

Error Conditions

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

Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    if (islower (ch))
        printf ("toupper=%#04x", toupper (ch));
    putchar ('\n');
}
```

See Also

islower, isupper, tolower

ungetc

Push character back into input stream

Synopsis

```
#include <stdio.h>
int ungetc(int uc, FILE *stream);
```

Description

The `ungetc` function pushes the character specified by `uc` back onto `stream`. The characters that have been pushed back onto `stream` will be returned by any subsequent read of `stream` in the reverse order of their pushing.

A successful call to the `ungetc` function will clear the `EOF` indicator for `stream`. The file position indicator for `stream` is decremented for every successful call to `ungetc`.

Upon successful completion, `ungetc` returns the character pushed back after conversion.

Error Conditions

If the `ungetc` function is unsuccessful, `EOF` is returned.

Example

```
#include <stdio.h>

void ungetc_example(FILE *fp)
{
   int ch, ret_ch;
   /* get char from file pointer */
   ch = fgetc(fp);
   /* unget the char, return value should be char */
```

```
    if ((ret_ch = ungetc(ch, fp)) != ch)
       printf("ungetc failed\n");
    /* make sure that the char had been placed in the file */
    if ((ret_ch = fgetc(fp)) != ch)
       printf("ungetc failed to put back the char\n");
}
```

See Also

fseek, fsetpos, getc

va_arg

Get next argument in variable-length list of arguments

Synopsis

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

Description

The `va_arg` macro is used to walk through the variable length list of arguments to a function.

After starting to process a variable-length list of arguments with `va_start`, call `va_arg` with the same `va_list` variable to extract arguments from the list. Each call to `va_arg` returns a new argument from the list.

Substitute a `type` name corresponding to the type of the next argument for the `type` parameter in each call to `va_arg`. After processing the list, call `va_end`.

The header file `stdarg.h` defines a pointer type called `va_list` that is used to access the list of variable arguments.

The function calling `va_arg` is responsible for determining the number and types of arguments in the list. It needs this information to determine how many times to call `va_arg` and what to pass for the type parameter each time. There are several common ways for a function to determine this type of information. The standard C `printf` function reads its first argument looking for %-sequences to determine the number and types of its extra arguments. In the example below, all of the arguments are of the same type (`char*`), and a termination value (NULL) is used to indicate the end of the argument list. Other methods are also possible.

If a call to va_arg is made after all arguments have been processed, or if va_arg is called with a type parameter that is different from the type of the next argument in the list, the behavior of va_arg is undefined.

Error Conditions

The va_arg macro does not return an error condition.

Example

```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>

char *concat(char *s1,...)
{
   int len = 0;
   char *result;
   char *s;
   va_list ap;

   va_start (ap,s1);
   s = s1;
   while (s){
      len += strlen (s);
      s = va_arg (ap,char *);
   }
   va_end (ap);

   result = malloc (len +7);
   if (!result)
      return result;
   *result = '\0';
   va_start (ap,s1);
   s = s1;
```

```
    while (s){
        strcat (result,s);
        s = va_arg (ap,char *);
    }
    va_end (ap);
    return result;
}

char *txt1 = "One";
char *txt2 = "Two";
char *txt3 = "Three";

extern int main(void)
{
    char *result;

    result = concat(txt1, txt2, txt3, NULL);

    puts(result);    /* prints "OneTwoThree" */
    free(result);
}
```

See Also

va_end, va_start

va_end

Finish variable-length argument list processing

Synopsis

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

Description

The `va_end` macro can only be invoked after the `va_start` macro has been invoked. A call to `va_end` concludes the processing of a variable-length list of arguments that was begun by `va_start`.

Error Conditions

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

Example

See "va_arg" on page 1-362

See Also

va_arg, va_start

va_start

Initialize the variable-length argument list processing

Synopsis

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

Description

The `va_start` macro is used to start processing variable arguments in a function declared to take a variable number of arguments. The first argument to `va_start` should be a variable of type `va_list`, which is used by `va_arg` to walk through the arguments.

The second argument is the name of the last *named* parameter in the function's parameter list; the list of variable arguments immediately follows this parameter. 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

See "va_arg" on page 1-362

See Also

va_arg, va_end

vfprintf

Print formatted output of a variable argument list

Synopsis

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

int vfprintf(FILE *stream, const char *format, va_list ap);
```

Description

The vfprintf function formats data according to the argument format, and then writes the output to the stream stream. The argument format contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to fprintf (on page 1-153) for a description of the valid format specifiers.

The vfprintf function behaves in the same manner as fprintf with the exception that instead of being a function which takes a variable number or arguments it is called with an argument list ap of type va_list, as defined in stdarg.h.

If the vfprintf function is successful, it will return the number of characters output.

Error Conditions

The vfprintf function returns a negative value if unsuccessful.

Example

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

void write_name_to_file(FILE *fp, char *name_template, ...)
```

```
{
   va_list p_vargs;
   int ret;                  /* return value from vfprintf */

   va_start (p_vargs,name_template);
   ret = vfprintf(fp, name_template, p_vargs);
   va_end (p_vargs);

   if (ret < 0)
      printf("vfprintf failed\n");
}
```

See Also

fprintf, va_start, va_end

vprintf

Print formatted output of a variable argument list to `stdout`

Synopsis

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

int vprintf(const char *format, va_list ap);
```

Description

The `vprintf` function formats data according to the argument `format`, and then writes the output to the standard output stream `stdout`. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to `fprintf` (on page 1-153) for a description of the valid format specifiers.

The `vprintf` function behaves in the same manner as `vfprintf` with `stdout` provided as the pointer to the stream.

If the `vprintf` function is successful it will return the number of characters output.

Error Conditions

The `vprintf` function returns a negative value if unsuccessful.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

void print_message(int error, char *format, ...)
```

```
{
   /* This function is called with the same arguments as for */
   /* printf but if the argument error is not zero, then the */
   /* output will be preceded by the text "ERROR:”           */

   va_list p_vargs;
   int ret;            /* return value from vprintf */

   va_start (p_vargs, format);
   if (!error)
      printf("ERROR: ");
   ret = vprintf(format, p_vargs);
   va_end (p_vargs);

   if (ret < 0)
      printf("vprintf failed\n");
}
```

See Also

fprintf, vfprintf

vsnprintf

Format argument list into an n-character array

Synopsis

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

int vsnprintf (char *str, size_t n, const char *format,
               va_list args);
```

Description

The `vsnprintf` function is similar to the `vsprintf` function in that it formats the variable argument list `args` according to the argument `format`, and then writes the output to the array `str`. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to `fprintf` (on page 1-153) for a description of the valid format specifiers.

The function differs from `vsprintf` in that no more than `n-1` characters are written to the output array. Any data written beyond the `n-1'th` character is discarded. A terminating `NUL` character is written after the end of the last character written to the output array unless `n` is set to zero, in which case nothing will be written to the output array and the output array may be represented by the `NULL` pointer.

The `vsnprintf` function returns the number of characters that would have been written to the output array `str` if `n` was sufficiently large. The return value does not include the terminating `NUL` character written to the array.

Error Conditions

The `vsnprintf` function returns a negative value if unsuccessful.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

char *message(char *format, ...)
{
   char *message = NULL;
   int len = 0;
   int r;
   va_list p_vargs;          /* return value from vsnprintf  */

   do {
      va_start (p_vargs,format);
      r = vsnprintf (message,len,format,p_vargs);
      va_end (p_vargs);
      if (r < 0)            /* formatting error?             */
         abort();
      if (r < len)          /* was complete string written?  */
         return message;    /* return with success           */
      message = realloc (message,(len=r+1));
   } while (message != NULL);
   abort();
}
```

See Also

fprintf, snprintf

vsprintf

Format argument list into a character array

Synopsis

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

int vsprintf (char *str, const char *format, va_list args);
```

Description

The `vsprintf` function formats the variable argument list `args` according to the argument `format`, and then writes the output to the array `str`. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to `fprintf` (on page 1-153) for a description of the valid format specifiers.

With one exception, the `vsprintf` function behaves in the same manner as `sprintf` with the exception that instead of being a function which takes a variable number of arguments, it is called with an argument list `args` of type `va_list`, as defined in `stdarg.h`.

The `vsprintf` function returns the number of characters that have been written to the output array `str`. The return value does not include the terminating `NUL` character written to the array.

Error Conditions

The `vsprintf` function returns a negative value if unsuccessful.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

char filename[128];

char *assign_filename(char *filename_template, ...)
{
   char *message = NULL;

   int r;
   va_list p_vargs;        /* return value from vsprintf */

   va_start (p_vargs,filename_template);
   r = vsprintf(&filename[0], filename_template, p_vargs);
   va_end (p_vargs);
   if (r < 0)              /* formatting error?         */
      abort();

   return &filename[0];    /* return with success       */
}
```

See Also

fprintf, sprintf, snprintf

write_extmem

Write to external memory

Synopsis

```
#include <21261.h>
#include <21262.h>
#include <21266.h>
#include <21267.h>
#include <21362.h>
#include <21363.h>
#include <21364.h>
#include <21365.h>
#include <21366.h>

void write_extmem(void    *internal_address,
                  void    *external_address,
                  size_t  n);
```

Description

On ADSP-2126x and some ADSP-2136x processors, it is not possible for the core to access external memory directly. The `write_extmem` function copies data from internal to external memory.

The `write_extmem` function will transfer `n` 32-bit words from `internal_address` to `external_address`.

Error Conditions

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

Example

See read_extmem for a usage example.

See Also

read_extmem

2 DSP RUN-TIME LIBRARY

This chapter describes the DSP run-time library, which contains a broad collection of functions that are commonly required by signal processing applications. The services provided by the DSP run-time library include support for general-purpose signal processing such as companders, filters, and Fast Fourier Transform (FFT) functions. These services are Analog Devices extensions to ANSI standard C.

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

The chapter contains the following:

- "DSP Run-Time Library Guide" on page 2-2 contains information about the library and provides a description of the DSP header files included with this release of the `cc21k` compiler.
- "DSP Run-Time Library Reference" on page 2-31 contains complete reference information for each DSP run-time library function included with this release of the `cc21k` compiler.

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:

Calling DSP Library Functions

To use a DSP run-time library function, call the function by name and provide the appropriate arguments. The names and arguments for each function are described in the function's reference page in "DSP Run-Time Library Guide" on page 2-2.

Like other functions you use, library functions should be declared. Declarations are supplied in header files, as described in "Working With Library Source Code" on page 2-5.

Note that C++ namespace prefixing is not supported when calling a DSP library function. All DSP library functions are in the C++ global namespace.

> The function names are C function names. If you call C run-time library functions from an assembly language program, you must use the assembly version of the function name, which is the function name prefixed with an underscore. For more information on naming conventions, see Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, section "C/C++ and Assembly Interface".

You can use the archiver, described in the *VisualDSP++ 5.0 Linker and Utilities Manual*, to build library archive files of your own functions.

Linking DSP Library Functions

When your C code calls a DSP run-time library function, the call creates a reference that the linker resolves when linking your program. 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 automatically directs the linker to the appropriate library under your VisualDSP++ installation. Table 2-1 lists the names of these libraries and where they are installed.

Table 2-1. DSP Run-Time Library File Names

Library Name	Directory	Processor
`libdsp020.dlb`	`21k\lib`	ADSP-21020 processors
`libdsp.dlb`	`21k\lib`	ADSP-2106X processors
`libdsp160.dlb`	`211xx\lib`	ADSP-2116x processors, built with `-workaround rframe,21161-anomaly-45`
`libdsp160.dlb`	`211xx\lib\swfa`	ADSP-2116x processors, built with `-workaround rframe,21161-anomaly-45,swfa`
`libdsp26x.dlb`	`212xx\lib`	ADSP-2126x processors
`libdsp26x.dlb`	`212xx\lib\2126x_rev_0.0`	ADSP-2126x processors, built with `-si-revision 0.0`

Table 2-1. DSP Run-Time Library File Names (Cont'd)

Library Name	Directory	Processor
`libdsp26x.dlb`	`212xx\lib\2126x_rev_any`	ADSP-2126x processors, built with `-si-revision any`
`libdsp36x.dlb`	`213xx\lib`	ADSP-213xx processors
`libdsp36x.dlb`	`213xx\lib\2136x_rev_0.0`	ADSP-2136x processors, built with `-si-revision 0.0`
`libdsp36x.dlb`	`213xx\lib\2136x_rev_any`	ADSP-2136x processors, built with `-si-revision any`
`libdsp37x.dlb`	`213xx\lib`	ADSP-2137x processors
`libdsp.dlb`	`214xx\lib`	ADSP-214xx processors
`libdsp.dlb`	`214xx\lib\21469_rev_any`	ADSP-214xx processors, built with `-si-revision any`
`libdsp_nwc.dlb`	`214xx\lib`	ADSP-214xx processors, built with `-nwc` (normal-word mode)
`libdsp_nwc.dlb`	`214xx\lib\21469_rev_any`	ADSP-214xx processors, built with `-si-revision any -nwc` (normal-word mode)

The library located in `212xx\lib` is built without any workarounds enabled; the library in `212xx\lib\212xx_rev_0.0` contains libraries that are suitable for revisions 0.0, 0.1, and 0.2; `212xx\lib\212xx_rev_any` contains libraries that will work with all revisions of ADSP-2126x processors.

The library located in `213xx\lib` is built without any workarounds enabled. The library in `213xx\lib\2136x_rev_0.0` contains libraries that are suitable for revisions 0.0, 0.1, and 0.2. The library in `213xx\lib\2136x_rev_any` contains libraries that will work with all revisions of ADSP-2136x processors.

The libraries located in `214xx\lib` are built without any workarounds enabled. In addition, a library directory called `21469_rev_any` is supplied. Libraries in this directory contain workarounds for all relevant anomalies on all revisions of ADSP-214xx processors.

If an application uses a customized Linker Description File, then either add the appropriate library to the `.ldf` file, or use the compiler's `-l` switch to add the appropriate DSP run-time library to the link-line. For example, `-ldsp37x` will add the library `libdsp37x.dlb` to the list of libraries to be searched by the linker. The `-l` switch is described in more detail in Chapter 1 of the manual VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors under the section "Compiler Command-Line Switches."

All the library functions in the DSP run-time library are re-entrant—they only operate on data passed in via a parameter and do not directly access non-constant static data. This means that the library may safely be used in a multi-threaded environment (such as VDK).

Library Attributes

The DSP run-time library contains the same attributes as the C/C++ run-time library. For more information, see "Library Attributes" in Chapter 1, C/C++ Run-Time Library.

Working With Library Source Code

The source code for the functions in the C and DSP run-time libraries is provided with VisualDSP++. By default, the source code is installed to a subdirectory of the directory where the run-time libraries are kept, named `<install_path>\21k\lib\src`, `<install_path>\211xx\lib\src`, `<install_path>\212xx\lib\src`, `<install_path>\213xx\lib\src`, and `<install_path>\214xx\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 startup 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 allows you to customize specific functions. To modify these files, you need proficiency in ADSP-21xxx assembly language and an understanding of the run-time environment, as explained in Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, section "C/C++ Run-Time Model and Environment".

Before modifying the source code, copy it to a file with a different filename and rename the function itself. Test the function before you use it in your system to verify that it is functionally correct.

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

DSP Header Files

The DSP header files contain prototypes for all the DSP library functions. When the appropriate `#include` preprocessor command is included in your source, the compiler uses the prototypes to check that each function is called with the correct arguments. Table 2-2 provides summaries of the DSP header files supplied with this release of the `cc21k` compiler.

Table 2-2. Summaries of DSP Header Files

Header File	Summary
"asm_sprt.h" on page 2-7	Mixed C/Assembly language macros
"cmatrix.h" on page 2-7	Arithmetic between complex matrices
"comm.h" on page 2-8	Scalar companders for A-law and μ-law
"complex.h" on page 2-8	Basic complex arithmetic functions
"cvector.h" on page 2-9	Arithmetic between complex vectors
"dma.h" on page 2-12	Functions for DMA operations
"filter.h" on page 2-12	Filters and transformations
"filters.h" on page 2-14	Filters operating on scalar input values
"macros.h" on page 2-15	Macros to access processor features

Table 2-2. Summaries of DSP Header Files (Cont'd)

Header File	Summary
"math.h" on page 2-15	Math functions
"matrix.h" on page 2-16	Matrix functions
"platform_include.h" on page 2-17	Platform-specific functions
"processor_include.h" on page 2-17	Processor-specific functions
"saturate.h" on page 2-19	Interface for saturated arithmetic operations
"sport.h" on page 2-19	Functions for ADSP-21xxx serial port
"stats.h" on page 2-19	Statistical functions
"sysreg.h" on page 2-19	Functions for access to SHARC system registers
"trans.h" on page 2-19	Fast Fourier Transform functions (not optimized for SHARC SIMD architectures)
"vector.h" on page 2-20	Vector functions
"window.h" on page 2-21	Window generators

The following sections describe the DSP header files in more detail.

asm_sprt.h

The `asm_sprt.h` header file consists of ADSP-21xxx assembly language macros, not C functions. They are used in your assembly routines that interface with C functions. For more information, see Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, section "Using Mixed C/C++ and Assembly Support Macros".

cmatrix.h

The `cmatrix.h` header file contains prototypes for functions that perform basic arithmetic between two complex matrices, and also between a complex matrix and a complex scalar. The supported complex types are described under the header file `complex.h`.

For a list of library functions that use this header, see Table 2-10 on page 2-26.

comm.h

The `comm.h` header file includes the voice-band compression and expansion communication functions that operate on scalar input values. However, the functions defined by this header file have not been optimized for the SHARC SIMD architectures.

A corresponding set of companding functions that operate on vectors and that have been optimized for the SHARC SIMD processors (that is, ADSP-211xx, ADSP-212xx, ADSP-213xx, and ADSP-214xx) are available in the header file `filter.h`.

> When compiling for a SHARC SIMD processor, the two different sets of companding functions defined in the `comm.h` and `filter.h` header files will have the same name but different parameters. Therefore, the user program should include the appropriate header file. The compiler will issue a fatal compilation error message if a source file being compiled for a SHARC SIMD processor includes both the `comm.h` and `filter.h` header files.

For a list of library functions that use this header, see Table 2-11 on page 2-26.

complex.h

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

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

```
typedef struct {
        float re;
```

```
        float im;
} complex_float;

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

typedef struct {
        long double re;
        long double im;
} complex_long_double;
```

Additional support for `complex numbers` is available via the `cmatrix.h` and `cvector.h` header files.

For a list of library functions that use this header, see Table 2-12 on page 2-26.

cvector.h

The `cvector.h` header file contains functions for basic arithmetic operations on vectors of type `complex_float`, `complex_double`, and `complex_long_double`. Support is provided for the dot product operation, as well as for adding, subtracting, and multiplying a vector by either a scalar or vector.

For a list of library functions that use this header, see Table 2-13 on page 2-27.

Header Files That Define Processor-Specific System Register Bits

The following header files define symbolic names for processor-specific system register bits. They also contain symbolic definitions for the IOP register address memory and IOP control/status register bits. Table 2-3 provides the header file names for processor-specific register bits.

Table 2-3. Header Files for Processor-Specific Register Bits

Header File	Processor
`def21020.h`	ADSP-21020 bit definitions
`def21060.h`	ADSP-21060 bit definitions
`def21061.h`	ADSP-21061 bit definitions
`def21062.h`	ADSP-21062 bit definitions
`def21065L.h`	ADSP-21065L bit definitions
`def21160.h`	ADSP-21160 bit definitions
`def21161.h`	ADSP-21161 bit definitions
`def21261.h`	ADSP-21261 bit definitions
`def21262.h`	ADSP-21262 bit definitions
`def21266.h`	ADSP-21266 bit definitions
`def21267.h`	ADSP-21267 bit definitions
`def21363.h`	ADSP-21363 bit definitions
`def21364.h`	ADSP-21364 bit definitions
`def21365.h`	ADSP-21365 bit definitions
`def21366.h`	ADSP-21366 bit definitions
`def21367.h`	ADSP-21367 bit definitions
`def21368.h`	ADSP-21368 bit definitions
`def21369.h`	ADSP-21369 bit definitions
`def21371.h`	ADSP-21371 bit definitions
`def21375.h`	ADSP-21375 bit definitions

Table 2-3. Header Files for Processor-Specific Register Bits (Cont'd)

Header File	Processor
`def21462.h`	ADSP-21462 bit definitions
`def21465.h`	ADSP-21465 bit definitions
`def21467.h`	ADSP-21467 bit definitions
`def21469.h`	ADSP-21469 bit definitions
`def21479.h`	ADSP-21479 bit definitions
`def21489.h`	ADSP-21489 bit definitions

Header Files That Allow Access to Memory-Mapped Registers From C/C++ Code

In order to allow safe access to memory-mapped registers from C/C++ code, the header files listed below are supplied. Each memory-mapped register's name is prefixed with "`p`" and is cast appropriately to ensure that the code is generated correctly. For example, `SYSCON` is defined as follows:

```
#define pSYSCON ((volatile unsigned int *) 0x00)
```

and can be used as:

```
*pSYSCON |= 0x6000;
```

Use this method of accessing memory-mapped registers in preference to using `asm` statements.

Supplied header files are:

Cdef21060.h	Cdef21061.h	Cdef21062.h	Cdef21065l.h
Cdef21160.h	Cdef21161.h	Cdef21261.h	Cdef21262.h
Cdef21266.h	Cdef21267.h	Cdef21363.h	Cdef21364.h
Cdef21365.h	Cdef21366.h	Cdef21367.h	Cdef21368.h
Cdef21369.h	Cdef21371.h	Cdef21375.h	Cdef21462.h
Cdef21465.h	Cdef21467.h	Cdef21469.h	Cdef21479.h
Cdef21489.h			

dma.h

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

filter.h

The `filter.h` header file contains filters and other key signal processing transformations such as Fast Fourier Transform (FFTs) and convolution. The header file also includes the A-law and µ-law companders that are used by voice-band compression and expansion applications.

The filters defined in this header file are finite and infinite impulse response filters, and multi-rate filters. All of these functions operate on an array of input samples; this is in contrast to the filter functions that are defined in `filters.h` and operate on scalars. Similarly, the A-law and µ-law companding functions of this header file input and output vectors whereas the companding functions of `comm.h` operate on one scalar at time.

The header file defines three different sets of FFT function. The first set is available when running on SHARC SIMD processors and includes the functions `cfftN`, `ifftN`, and `rfftN`, where `N` stands for the size of FFT computed (that is `N` represents 16, 32, 64 ...). These functions are relatively slow but they require the least amount of code memory, and the least amount of data memory as they re-use the input array as temporary storage. Each of the FFT functions includes an internal twiddle table (which is a set of sine and cosine coefficients required by FFT functions) that has been tailored to the explicit size of the FFT being generated. For example, the functions `cfft32`, `ifft32`, and `rfft32` share one twiddle table and `cfft64`, `ifft64`, and `rfft64` share another. The size of each twiddle table is `FFTSIZE` words and is allocated in DM memory. Therefore the advantages of smaller code size and data size diminishes if an application calculates FFTs of more than one size as it will include a set of FFT functions and associated twiddle tables for each of the different sizes of FFT computed.

The second set of Fast Fourier Transform functions is defined for all SHARC processors and is composed of the functions `cfft`, `ifft`, and `rfft`. The number of points in the FFT is passed as a parameter to these functions. The functions are also passed a twiddle table which can be shared if an application calculates FFTs that have more than one size. These functions have the ability to preserve the input data. Their memory footprint is larger than the first set of FFT functions, but an application will only include one instance of an FFT function no matter how many different-sized FFTs that it calculates. These FFT functions are faster and more flexible than the first set, and will be more memory efficient if an application calculates FFTs of different sizes.

The third set of FFT functions that is defined by this header file represent a set of highly optimized functions that are only available on the ADSP-21xxx SIMD platforms. This set of functions, represented by `cfftf`, `ifftf`, and `rfftf_2`, sacrifice a level of flexibility in favor of optimal performance. For example, twiddle tables cannot be shared when computing FFTs of different sizes and these FFT functions overwrite the input data. The input arrays must be aligned on an address boundary that is a multiple of the FFT size, and the functions cannot be used to reference external memory. Memory usage lies between the first and second set of FFT functions.

The `trans.h` header file defines a set of alternative set of FFT functions that are supported on all SHARC processors, but these functions have not been optimized for the SIMD architectures of the ADSP-211xx, ADSP-212xx, ADSP-213xx, and ADSP-214xx family of processors.

The header file also defines library functions that compute the magnitude of an FFT, and a function that convolves two arrays.

> If a source file compiled for a SHARC SIMD processor includes the `filter.h` header file, then it must not also include one of the header files `comm.h`, `filters.h`, or `trans.h`. Any attempt to do so causes the compiler to issue a fatal compilation error.

For a list of library functions that use this header, see Table 2-16 on page 2-27.

filters.h

The `filters.h` header file includes finite and infinite impulse response filters that operate on scalar input values. However, the functions defined by this header file are not optimized for the ADSP-211xx/212xx/213xx/214xx SIMD architectures.

Note that alternative filter functions that operate on vectors are defined in the `filter.h` header file. These functions will also exploit the SIMD capabilities of the ADSP-211xx, ADSP-212xx, ADSP-213xx, and ADSP-214xx processors.

> When compiling for a SHARC SIMD processor, the two different sets of filter functions that are defined in the `filter.h` and `filters.h` header files will have the same name but different parameters. It is important therefore that the user program includes the appropriate header file. The compiler will issue a fatal compilation error message if a source file is being compiled for a SHARC SIMD processors and it includes both the `filter.h` and `filters.h` header files.

For a list of library functions that use this header, see Table 2-15 on page 2-27.

macros.h

The `macros.h` header file contains a collection of macros and other definitions that allow some access to special computational features of the underlying hardware. Some portions of this file are present for compatibility with previous releases of the VisualDSP++ toolset. In these cases, newer implementations provide equal or better access to the underlying functionality.

math.h

The standard math functions defined in the `math.h` header file have been augmented by implementations for the `float` and `long double` data types and additional functions that are Analog Devices extensions to the ANSI standard.

Table 2-4 provides a summary of the additional library functions defined by the `math.h` header file.

Table 2-4. Math Library – Additional Functions

Description	Prototype

Anti-log	`double alog (double x);`
	`float alogf (float x);`
	`long double alogd (long double x);`

Average	`double favg (double x, double y);`
	`float favgf (float x, float y);`
	`long double favgd (long double x, long double y);`

Base 10 anti-log	`double alog10 (double x);`
	`float alog10f (float x);`
	`long double alog10d (long double x);`

Table 2-4. Math Library – Additional Functions (Cont'd)

Description	Prototype

Clip	`double fclip (double x, double y);`
	`float fclipf (float x, float y);`
	`long double fclipd (long double x, long double y);`

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

Detect Infinity	`int isinf (double x);`
	`int isinff (float x);`
	`int isinfd (long double x);`

Detect NaN	`int isnan (double x);`
	`int isnanf (float x);`
	`int isnand (long double x);`

Maximum	`double fmax (double x, double y);`
	`float fmaxf (float x, float y);`
	`long double fmaxd (long double x, long double y);`

Minimum	`double fmin (double x, double y);`
	`float fminf (float x, float y);`
	`long double fmind (long double x, long double y);`

Reciprocal of square root	`double rsqrt (double x);`
	`float rsqrtf (float x);`
	`long double rsqrtd (long double x);`

Sign copy	`double copysign (double x, double y);`
	`float copysignf (float x, float y);`
	`long double copysignd (long double x, long double y);`

For a list of library functions that use this header, see Table 2-17 on page 2-28.

matrix.h

The `matrix.h` header file declares a number of function prototypes associated with basic arithmetic operations on matrices of type `float`, `double`, and `long double`. The header file contains support for arithmetic between two matrices, and between a matrix and a scalar.

For a list of library functions that use this header, see Table 2-18 on page 2-28.

platform_include.h

The `platform_include.h` header file includes the appropriate header files that define symbolic names for processor-specific system register bits. These header files also contain symbolic definitions for the IOP register address memory and IOP control/status register bits. With the exception of ADSP-21020, `platform_include.h` causes 1 or 2 include files to be included, depending on whether assembly or C/C++ code is being processed.

For more information on the platform-specific include files, see the following sections:

- "Header Files That Define Processor-Specific System Register Bits" on page 2-10
- "Header Files That Allow Access to Memory-Mapped Registers From C/C++ Code" on page 2-11

processor_include.h

The `processor_include.h` header file includes the appropriate header file that defines the processor-specific functions of the DSP run-time library, such as `poll_flag_in()` and `idle()`. The processor header file also includes support for initializing, enabling, and disabling the processor's programmable timer (or, in the case of the ADSP-21065L processor, the processor's two programmable timers). The `processor_include.h` header file will include one of the header files found in Table 2-5, depending upon the target processor.

Table 2-5. Processor-Specific Header Files

Header File	Header File Processor-Specific Content
21020.h	ADSP-21020 DSP functions
21060.h	ADSP-2106x DSP functions
21065l.h	ADSP-21065L DSP functions
21160.h	ADSP-21160 DSP functions
21161.h	ADSP-21161 DSP functions
21261.h	ADSP-21261 DSP functions
21262.h	ADSP-21262 DSP functions
21266.h	ADSP-21266 DSP functions
21267.h	ADSP-21267 DSP functions
21363.h	ADSP-21363 DSP functions
21364.h	ADSP-21364 DSP functions
21365.h	ADSP-21365 DSP functions
21366.h	ADSP-21366 DSP functions
21367.h	ADSP-21367 DSP functions
21368.h	ADSP-21368 DSP functions
21369.h	ADSP-21369 DSP functions
21371.h	ADSP-21371 DSP functions
21375.h	ADSP-21375 DSP functions
21462.h	ADSP-21462 DSP functions
21465.h	ADSP-21465 DSP functions
21467.h	ADSP-21467 DSP functions
21469.h	ADSP-21469 DSP functions
21479.h	ADSP-21479 DSP functions
21489.h	ADSP-21489 DSP functions

For a list of library functions that use this header, see Table 2-19 on page 2-29.

saturate.h

The `saturate.h` header file defines the interface for saturated arithmetic operations. See Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, section “Saturated Arithmetic” for further information.

sport.h

The `sport.h` header file provides definitions and setup, enable, and disable functions for the ADSP-21xxx DSP serial ports.

stats.h

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

For a list of library functions that use this header, see Table 2-20 on page 2-29.

sysreg.h

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 Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, section “Access to System Registers”.

trans.h

The `trans.h` header file includes the Fast Fourier Transform (FFT) functions. These functions operate on data in which the real and imaginary parts of the input and output signal are stored in separate vectors. They have not be optimized for the ADSP-21xxx SIMD architectures. The header file defines the functions `cfftN`, `ifftN`, and `rfftN`, where `N` stands for the number of points that the FFT function will compute (that is 16, 32, 64, ...).

The `cfftN` and `ifftN` functions respectively compute the FFT, and inverse FFT, from an N-point complex input signal. The `rfftN` functions are similar to the `cfftN` functions, except that they operate on input signals of real data only; this is equivalent to `cfftN` whose imaginary input component is set to zero.

Alternative FFTs functions, that have been optimized for the ADSP-21xxx SIMD processors, are defined in the `filter.h` header file.

> When compiling for a SHARC SIMD processor, the two different sets of FFT functions that are defined in the `trans.h` and `filter.h` header files will have the same name but different parameters. It is important therefore that the user program includes the appropriate header file. The compiler will issue a fatal compilation error message if a source file is being compiled for a SHARC SIMD processors and it includes both the `trans.h` and `filter.h` header files.

For a list of library functions that use this header, see Table 2-21 on page 2-29.

vector.h

The `vector.h` header file contains functions for operating on vectors of type `float`, `double`, and `long double`. Support is provided for the dot product operation as well as for adding, subtracting, and multiplying a vector by either a scalar or vector. Similar support for the complex data types is defined in the header file `cvector.h`.

For a list of library functions that use this header, see Table 2-22 on page 2-30.

window.h

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 2-6.

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 2-6. 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)`
Generate von Hann window	`void gen_vonhann (float w[], int a, int n)`

For a list of library functions that use this header, see Table 2-23 on page 2-30.

Built-In DSP Library 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 by the cc21k compiler are listed in Table 2-7. Refer to "Using Compiler Built-In C Library Functions" on page 1-39 for more information on this topic.

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

Table 2-7. Built-in DSP Functions

`avg`	`clip`	`copysign`	`copysignf`
`favg`	`favgf`	`fmax`	`fmaxf`
`fmin`	`fminf`	`labs`	`lavg`
`lclip`	`lmax`	`lmin`	`max`
`min`			

> Functions `copysign`, `favg`, `fmax`, and `fmin` are compiled as a built-in function only if `double` is the same size as `float`.

The compiler also supports a set of built-in functions for which no inline 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 generates a call to an appropriate version of the run-time library function. Table 2-8 lists library functions that may be called with pointers that address Program Memory.

Table 2-8. Library Functions Called With Pointers

`histogram`	`matmaddf`	`matmmltf`
`matmsubf`	`matsaddf`	`matsmltf`
`matssubf`	`meanf`	`rmsf`
`transpmf`	`varf`	`zero_crossf`

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

Implications of Using SIMD Mode

The ADSP-2116x, ADSP-2126x, ADSP-213xx, and ADSP-214xx processors support Single-Instruction, Multiple-Data (SIMD) operations, which, under certain conditions, double the computational rate over ADSP-2106x processors. The DSP run-time library for these processors makes extensive use of their SIMD capabilities. In essence, when running in SIMD mode, data contained in memory is always accessed as two 32-bit words, starting at an even word boundary. Therefore, it is essential that any array that is passed to a DSP library function be allocated on a double-word (even word) boundary.

The `cc21k` compiler normally aligns arrays properly in memory. However, the compiler cannot control the allocation of all arrays that are used as arguments to DSP library functions. For example, the alignment of the array `&a[i]` is controlled by the value of the scalar `i`. If the value of the scalar is odd, then the library function might return incorrect results. A variant of this example involves the use of pointers to arrays. If the variable `ptr` is initialized using `ptr=&a[i]` and the value of the scalar `i` is odd, then you cannot use `ptr` to pass an array to a DSP library function

Refer to Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, section “Restrictions to Using SIMD” for more information on this topic. The SIMD feature is described in detail in the same chapter, in the section entitled “SIMD Support”.

A limited number of DSP library functions, whose arguments involve the use of arrays, do not use the SIMD feature of ADSP-2116x, ADSP-2126x, ADSP-213xx, and ADSP-214xx processors due to the nature of their algorithm. These library functions include all `long double` functions and the window generators. In addition, Table 2-9 lists the following functions:

Table 2-9. Functions Not Using the SIMD Feature

biquad	cmatmmlt	cmatsmlt	convolve
cvecdot	cvecsmlt	fir_decima	fir_interp
iir	histogram	matmmlt	matinv
transpm	zero_cross		

Some ADSP-2116x, ADSP-2126x, and ADSP-213xx processors have restrictions on the use of SIMD access to data placed in external memory. For more information, see “Using Data in External Memory” on page 2-24.

Using Data in External Memory

The run-time functions described in this manual have been optimized to exploit the features of the SHARC architecture. This can lead to restrictions in the placement of data in external memory, particularly on some ADSP-211xx, ADSP-212xx, and ADSP-213xx processors.

The ADSP-212xx and some ADSP-2136x processors do not support direct memory accesses to external memory. This means that the run-time functions cannot read or write to data in external memory. Any such data must first be brought into internal memory. The library functions `read_extmem` and `write_extmem` may be used to transfer data between internal memory and external memory.

Some ADSP-211xx and ADSP-213xx processors have a 32-bit external bus and, due to the shorter bus width, are unable to support SIMD access to external memory. For this reason, the DSP library contains an alternative set of functions that do not use the architecture's SIMD capabilities. This alternative set is selected in preference to the standard library functions if the `-no-simd` compiler switch is specified at compilation time.

The ADSP-214xx processors do support SIMD access to external memory, but not long word (LW) access to external memory. Therefore the `cvecvmltf` library function is not suitable for use with data placed in external memory, since it makes use of the LW mnemonic. (This also applies to the `cvecvmlt` function if doubles are the same size as floats.) An alternative version of the function does not use the architecture's SIMD capabilities and is suitable for use with data placed in external memory. This version is available by way of the `-no-simd` compiler switch.

The optimized FFT functions `cfftf`, `ifftf`, and `rfftf_2` use both SIMD and long word memory accesses to improve their performance. All data passed to these functions must be allocated in internal memory. There are no versions of these functions that support data in external memory.

Documented Library Functions

The C run-time library has several categories of functions and macros defined by the ANSI C standard, plus extensions provided by Analog Devices.

The following tables list the library functions documented in this chapter. Note that the tables list the functions for each header file separately; however, the reference pages for these library functions present the functions in alphabetical order.

Table 2-10 lists the library functions in the `cmatrix.h` header file. Refer to "cmatrix.h" on page 2-7 for more information on this header file.

Table 2-10. Library Functions in `cmatrix.h`

cmatmadd	cmatmmlt	cmatmsub
cmatsadd	cmatsmlt	cmatssub

Table 2-11 lists the library functions in the `comm.h` header file. Refer to "comm.h" on page 2-8 for more information on this header file.

Table 2-11. Library Functions in `comm.h`

a_compress	a_expand	mu_compress
mu_expand		

Table 2-12 lists the library functions in the `complex.h` header file. Refer to "complex.h" on page 2-8 for more information on this header file.

Table 2-12. Supported Library Functions in `complex.h`

arg	cabs	cadd
cartesian	cdiv	cexp
cmlt	conj	csub
norm	polar	

Table 2-13 lists the library functions in the `cvector.h` header file. Refer to "cvector.h" on page 2-9 for more information on this header file.

Table 2-13. Supported Library Functions in `cvector.h`

cvecdot	cvecsadd	cvecsmlt
cvecssub	cvecvadd	cvecvmlt
cvecvsub		

Table 2-14 lists the library functions in the `dma.h` header file. Refer to "dma.h" on page 2-12 for more information on this header file.

Table 2-14. Supported Library Functions in `dma.h`

dma_disable	dma_enable	dma_setup
dma_status		

Table 2-15 lists the library functions in the `filters.h` header file. Refer to "filters.h" on page 2-14 for more information on this header file.

Table 2-15. Supported Library Functions in `filters.h`

biquad	fir	iir

Table 2-16 lists the library functions in the `filter.h` header file. Refer to "filter.h" on page 2-12 for more information on this header file.

Table 2-16. Supported Library Functions in `filter.h`

a_compress	a_expand	biquad
cfft	cfft_mag (SHARC SIMD Processors)	cfftN (SHARC SIMD Processors)
cfftf (SHARC SIMD Processors)	convolve	fft_magnitude

Table 2-16. Supported Library Functions in `filter.h` (Cont'd)

fftf_magnitude (SHARC SIMD Processors)	fir	fir_decima
fir_interp	ifft	ifftf (SHARC SIMD Processors)
ifftN (SHARC SIMD Processors)	iir	mu_compress
mu_expand	rfft	rfft_mag (SHARC SIMD Processors)
rfftf_2 (SHARC SIMD Processors)	rfftN (SHARC SIMD Processors)	twidfft
twidfftf (SHARC SIMD Processors)		

Table 2-17 lists the library functions in the `math.h` header file. Refer to "math.h" on page 2-15 for more information on this header file.

Table 2-17. Supported Library Functions in `math.h`

alog	alog10	copysign
cot	favg	fclip
fmax	fmin	rsqrt

Table 2-18 lists the library functions in the `matrix.h` header file. Refer to "matrix.h" on page 2-16 for more information on this header file.

Table 2-18. Supported Library Functions in `matrix.h`

matinv	matmadd	matmmlt
matmsub	matsadd	matsmlt
matssub	transpm	

Table 2-19 lists the library functions in the `processor_include.h` header file. Refer to "processor_include.h" on page 2-17 for more information on this header file.

Table 2-19. Library Functions in `processor_include.h`

circindex	circptr	idle
poll_flag_in	set_flag	set_semaphore
test_and_set_semaphore	timer_off	timer_on
timer0_off, timer1_off (ADSP-21065L Processor Only)		

Table 2-20 lists the library functions in the `stats.h` header file. Refer to "stats.h" on page 2-19 for more information on this header file.

Table 2-20. Supported Library Functions in `stats.h`

autocoh	autocorr	crosscoh
crosscorr	histogram	mean
rms	var	zero_cross

Table 2-21 lists the library functions in the `trans.h` header file. Refer to "trans.h" on page 2-19 for more information on this header file.

Table 2-21. Supported Library Functions in `trans.h`

cfftN	ifftN	rfftN

Table 2-22 lists the library functions in the `vector.h` header file. Refer to "vector.h" on page 2-20 for more information on this header file.

Table 2-22. Supported Library Functions in `vector.h`

vecdot	vecsadd	vecsmlt
vecssub	vecvadd	vecvmlt
vecvsub		

Table 2-23 lists the library functions in the `window.h` header file. Refer to "window.h" on page 2-21 for more information on this header file.

Table 2-23. Supported Library Functions in `window.h`

gen_bartlett	gen_blackman	gen_gaussian
gen_hamming	gen_hanning	gen_harris
gen_kaiser	gen_rectangular	gen_triangle
gen_vonhann		

DSP Run-Time Library Reference

The DSP run-time library is a collection of functions that you can call from your C/C++ programs. This section lists the functions in alphabetical order, for both ADSP-21xxx SIMD and ADSP-210xx processors. Functions that apply to only one processor family are labeled as such. 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.

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 have the following format:

Name and purpose of the function

Synopsis – Required header file and functional prototype

Description – Function specification

Algorithm – High-level mathematical representation of the function

Error Conditions – Method that the functions use to indicate an error

Example – Typical function usage

See Also – Related functions

a_compress

A-law compression

Synopsis (Scalar-Valued Version)

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

Synopsis (Vector-Valued Version)

ADSP-210xx Processors

```
#include <filter.h>

int *a_compress_vec (const int dm input[],
                int dm output[],
                int length);
```

ADSP-21xxx SIMD Processors

```
#include <filter.h>

int *a_compress (const int dm input[],
                int dm output[],
                int length);
```

Description

The A-law compression functions take a linear 13-bit signed speech sample and compresses it according to ITU recommendation G.711.

The scalar-valued version of `a_compress` inputs a single data sample and returns an 8-bit compressed output sample.

The vector-valued version of `a_compress` takes the array `input`, and returns the compressed 8-bit samples in the vector `output`. The parameter `length` defines the size of both the input and output vectors. The function returns a pointer to the output array.

The vector-valued version of `a_compress` uses serial port 0 to perform the companding on an ADSP-21160 processor; serial port 0 therefore must not be in use when this routine is called. The serial port is not used by this function on any other ADSP-21xxx SIMD architectures.

Error Conditions

The A-law compression functions do not return an error condition.

Example

Scalar-Valued

```
#include <comm.h>

int sample, compress;
compress = a_compress (sample);
```

Vector-Valued

```
#include <filter.h>

#define NSAMPLES 50
int data[NSAMPLES], compressed[NSAMPLES];
#if defined(__SIMDSHARC__)
   a_compress (data, compressed, NSAMPLES);
#else
   a_compress_vec (data, compressed, NSAMPLES);
#endif
```

See Also

a_expand, mu_compress

a_expand

A-law expansion

Synopsis (Scalar-Valued Version)

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

Synopsis (Vector-Valued Version)

ADSP-210xx Processors

```
#include <filter.h>

int *a_expand_vec (const int dm input[],
                int dm output[],
                int length);
```

ADSP-21xxx SIMD Processors

```
#include <filter.h>

int *a_expand (const int dm input[],
                int dm output[],
                int length);
```

Description

The `a_expand` function takes an 8-bit compressed speech sample and expands it according to ITU recommendation G.711 (A-law definition).

The scalar version of `a_expand` inputs a single data sample and returns a linear 13-bit signed sample.

The vector version of the `a_expand` function takes an array of 8-bit compressed speech samples and expands them according to ITU recommendation G.711 (A-law definition). The array returned contains linear 13-bit signed samples. This function returns a pointer to the `output` data array.

> The vector version of the `a_expand` function uses serial port 0 to perform the companding on an ADSP-21160 processor; serial port 0 therefore must not be in use when this routine is called. The serial port is not used by this function on any other ADSP-21xxx SIMD architectures.

Error Conditions

The A-law expansion functions do not return an error condition.

Example

Scalar-Valued

```
#include <comm.h>

int compressed_sample, expanded;
expanded = a_expand (compressed_sample);
```

Vector-Valued

```
#include <filter.h>

#define NSAMPLES 50
int compressed_data[NSAMPLES];
int expanded_data[NSAMPLES];

#if defined(__SIMDSHARC__)
    a_expand (compressed_data, expanded_data, NSAMPLES);
#else
    a_expand_vec (compressed_data, expanded_data, NSAMPLES);
#endif
```

See Also

a_compress, mu_expand

alog

Anti-log

Synopsis

```
#include <math.h>

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

Description

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

Error Conditions

The input argument `x` for `alogf` must be in the domain [-87.3, 88.7] and the input argument for `alogd` must be in the domain [-708.2, 709.1]. The functions return `HUGE_VAL` if `x` is greater than the domain, and return 0.0 if `x` is less than the domain.

Example

```
#include <math.h>

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

See Also

alog10, exp, log, pow

alog10

Base 10 anti-log

Synopsis

```
#include <math.h>

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

Description

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

Error Conditions

The input argument `x` for `alog10f` must be in the domain [-37.9, 38.5], and the input argument for `alog10d` must be in the domain [-307.57, 308.23]. The functions return `HUGE_VAL` if `x` is greater than the domain, and they return 0.0 if `x` is less than the domain.

Example

```
#include <math.h>

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

See Also

alog, exp, log10, pow

arg

Get phase of a complex number

Synopsis

```
#include <complex.h>

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

Description

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

Algorithm

The phase of a Cartesian number is computed as:

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

Error Conditions

The `arg` functions return a zero if `a.re` <> 0 and `a.im` = 0.

Example

```
#include <complex.h>

complex_float x = {0.0,1.0};
float r;
r = argf(x);      /* r = pi/2 */
```

See Also

atan2, cartesian, polar

autocoh

Autocoherence

Synopsis

```
#include <stats.h>

float *autocohf (float dm out[],
                 const float dm in[],
                 int samples, int lags);

double *autocoh (double dm out[],
                 const double dm in[],
                 int samples, int lags);

long double *autocohd (long double dm out[],
                       const long double dm in[],
                       int samples, int lags);
```

Description

The autocoherence functions compute the autocoherence of the floating-point input, `in[]`, which contain `samples` values. The autocoherence of an input signal is its autocorrelation minus its mean squared. The functions return a pointer to the output array `out[]` of length `lags`.

For the ADSP-21xxx SIMD processors the `autocohf` function (and `autocoh`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

Algorithm

The following equation is the basis of the algorithm.

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

where:

k = {0, 1, ..., `lags`-1}

$\bar{a}$ is the mean value of input vector `a`

Error Conditions

The autocoherence functions do not return an error condition.

Example

```
#include <stats.h>
#define SAMPLES 1024
#define LAGS      16

double excitation[SAMPLES];
double response[LAGS];
int lags = LAGS;

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

See Also

autocorr, crosscoh, crosscorr

autocorr

Autocorrelation

Synopsis

```
#include <stats.h>

float *autocorrf (float dm out[], const float dm in[],
                  int samples, int lags);

double *autocorr (double dm out[], const double dm in[],
                  int samples, int lags);

long double *autocorrd (long double dm out[],
                        const long double dm in[],
                        int samples, int lags);
```

Description

The autocorrelation functions perform an autocorrelation of a signal. Autocorrelation is the cross-correlation of a signal with a copy of itself. It provides information about the time variation of the signal. The signal to be autocorrelated is given by the `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`. The functions return a pointer to the `out[]` output data array of length `lags`.

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

For the ADSP-21xxx SIMD processors the `autocorrf` function (and `autocorr`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

Algorithm

The following equation is the basis of the algorithm.

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

where:

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

Error Conditions

The autocorrelation functions do not return an error condition.

Example

```
#include <stats.h>
#define SAMPLES 1024
#define LAGS      16

double excitation[SAMPLES];
double response[LAGS];
int lags = LAGS;

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

See Also

autocoh, crosscoh, crosscorr

biquad

Biquad filter section

Synopsis (Scalar-Valued Version)

```
#include <filters.h>

float biquad (float                sample,
              const float    pm coeffs[],
              float          dm state[],
              int               sections);
```

Synopsis (Vector-Valued Version)

ADSP-210xx Processors

```
#include <filter.h>

float *biquad_vec (const float  dm input[],
                   float         dm output[],
                   const float   pm coeffs[],
                   float         dm state[],
                   int             samples,
                   int             sections);
```

ADSP-21xxx SIMD Processors

```
#include <filter.h>

float *biquad (const float   dm input[],
               float          dm output[],
               const float    pm coeffs[],
               float          dm state[],
               int              samples,
               int              sections);
```

Description

The `biquad` functions implement a cascaded biquad filter defined by the coefficients and the number of sections that are supplied in the call to the function.

The scalar version of `biquad` produces the filtered response of its input data `sample` which it returns as the result of the function.

The vector versions of the `biquad` function generate the filtered response of the input data `input` and store the result in the output vector `output`. The number of input samples and the length of the output vector is specified by the argument `samples`.

The number of biquad sections is specified by the parameter `sections`, and each biquad section is represented by five coefficients `A1`, `A2`, `B0`, `B1`, and `B2`. The biquad functions assume that the value of `A0` is `1.0`, and `A1` and `A2` should be scaled accordingly. These coefficients are passed to the biquad functions in the array `coeffs` which must be located in Program Memory (PM). The definition of the `coeffs` array is:

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

For the scalar version of `biquad` the five coefficients of each section must be stored in reverse order:

```
B2, B1, B0, A2, A1
```

For the vector versions of the `biquad` function, the five coefficients must be stored in the order:

```
A2, A1, B2, B1, B0
```

When importing coefficients from most filter design tools, the `A1` and `A2` coefficients should be negated.

Each filter should have its own delay line, which is represented by the array `state`. The `state` array should be large enough for two delay elements per biquad section and hold an internal pointer that allows the filter to be restarted.

The definition of the state is:

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

The `state` array should be initially cleared to zero before calling the function for the first time, and should not otherwise be modified by the user program.

The library function uses the architecture's dual-data move instruction to provide simultaneous access to the filter coefficients (in PM data memory) and the delay line. When running on an ADSP-21367, ADSP-21368, ADSP-21369, ADSP-21371, or ADSP-21375 processor, the filter coefficients and the delay line must not both be allocated in external memory; otherwise, the function can generate an incorrect set of results. This occurs because in a dual-data move instruction, the hardware does not support both memory accesses allocated to external memory. Therefore, ensure that the filter coefficients or the delay line (or, optionally, both) are allocated in internal memory when running on one of the 213xx processors specified above.

The vector version of the `biquad` functions return a pointer to the output vector; the scalar version of the function returns the filtered response of its input sample.

Algorithm

The following equations are the basis of the algorithm.

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

where

$$D_m = A_2 \bullet D_{m-2} + A_1 \bullet D_{m-1} + x_m$$

where m = {0,1,2,...,`samples`-1}

$$Y_m = B_2 \bullet D_{m-2} + B_1 \bullet D_{m-1} + B_0 \bullet D_m$$

The algorithm used is adapted from *Digital Signal Processing*, Oppenheim and Schafer, New Jersey, Prentice Hall, 1975. For more information, see Figure 2-1 on page 2-51.

Error Conditions

The `biquad` functions do not return an error condition.

Example

Scalar-Valued

```
#include <filters.h>

#define NSECTIONS  4
#define NSTATE    ((2*NSECTIONS) + 1)

float sample, response, state[NSTATE];
float pm coeffs[5*NSECTIONS];
int i;

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

response = biquad (sample, coeffs, state, NSECTIONS);
```

Vector-Valued

```
#include <filter.h>

#define NSECTIONS  4
#define NSAMPLES  64
#define NSTATE    ((2*NSECTIONS) + 1)

float input[NSAMPLES];
float output[NSAMPLES];

float state[NSTATE];
float pm coeffs[5*NSECTIONS];
int i;
```

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

#if defined(__SIMDSHARC__)
    biquad (input, output, coeffs, state, NSAMPLES,
            NSECTIONS);
#else
    biquad_vec (input, output, coeffs, state,
            NSAMPLES, NSECTIONS);
#endif
```

See Also

fir, iir

The `biquad` function makes use of circular buffers. Refer to the "Interrupts and Circular Buffering" section of Chapter 1, "Compiler", in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* for information on interrupt dispatcher considerations when circular buffers are used within an application.

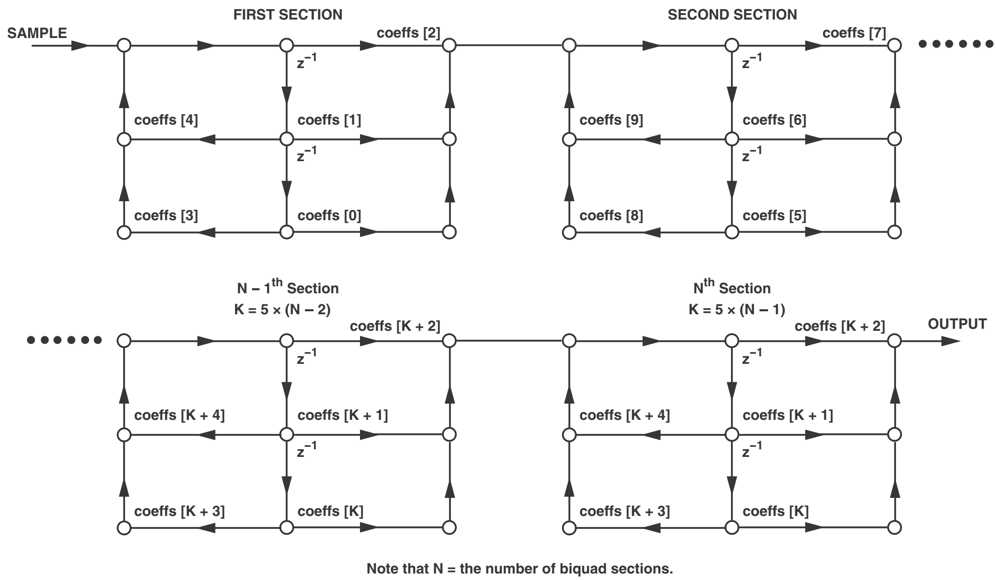

Figure 2-1. Biquad Sections

cabs

Complex absolute value

Synopsis

```
#include <complex.h>

float cabsf (complex_float z);
double cabs (complex_double z);
long double cabsd (complex_long_double z);
```

Description

The `cabs` functions return the floating-point absolute value of their complex input.

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

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

Error Conditions

The `cabs` functions do 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, labs

cadd

Complex addition

Synopsis

```
#include <complex.h>

complex_float caddf (complex_float a, complex_float b);
complex_double cadd (complex_double a, complex_double b);
complex_long_double caddd (complex_long_double a,
                           complex_long_double b);
```

Description

The `cadd` functions add the two complex values `a` and `b` together, and return the result.

Error Conditions

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

Example

```
#include <complex.h>

complex_double x = {9.0,16.0};
complex_double y = {1.0,-1.0};
complex_double z;

z = cadd (x,y);      /* z.re = 10.0, z.im = 15.0 */
```

See Also

cdiv, cmlt, csub

cartesian

Convert Cartesian to polar notation

Synopsis

```
#include <complex.h>

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

Description

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

The formula for converting from Cartesian to polar notation is given by:

```
magnitude = cabs(a)

phase = arg(a)
```

Error Conditions

The `cartesian` functions return a zero for the phase if `a.re` <> 0 and `a.im` = 0.

Example

```
#include <complex.h>

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

See Also

arg, cabs, polar

cdiv

Complex division

Synopsis

```
#include <complex.h>

complex_float cdivf (complex_float a, complex_float b);
complex_double cdiv (complex_double a, complex_double b);
complex_long_double cdivd (complex_long_double a,
                           complex_long_double b);
```

Description

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

Algorithm

The following equation is the basis of the algorithm.

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

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

Error Conditions

The `cdiv` functions set both the real and imaginary parts of the result to Infinity if `b` is equal to (`0.0,0.0`).

Example

```
#include <complex.h>

complex_double x = {3.0,11.0};
complex_double y = {1.0, 2.0};
complex_double z;

z = cdiv (x,y);      /* z.re = 5.0, z.im = 1.0 */
```

See Also

cadd, cmlt, csub

cexp

Complex exponential

Synopsis

```
#include <complex.h>

complex_float cexpf (complex_float z);
complex_double cexp (complex_double z);
complex_long_double cexpd (complex_long_double z);
```

Description

The `cexp` functions compute the exponential value `e` to the power of the real argument `z` in the complex domain. The exponential of a complex value is evaluated with the following formula.

```
Re(y) = exp (Re(z)) * cos (Im(z));

Im(y) = exp (Re(z)) * sin (Im(z));
```

Error Conditions

For underflow errors, the `cexp` functions return 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

log, pow

cfft

Complex radix-2 Fast Fourier Transform

Synopsis

```
#include <filter.h>

complex_float  *cfft (complex_float        dm input[],
                      complex_float        dm temp[],
                      complex_float        dm output[],
                      const complex_float  pm twiddle[],
                      int                  twiddle_stride,
                      int                  n);
```

Description

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

The size of the input array `input`, the output array `output`, and the temporary working buffer `temp` must be at least `n`, where `n` represents the number of points in the FFT; `n` must be a power of 2 and no smaller than 8. If the input data can be overwritten, memory can be saved by setting the pointer of the temporary array explicitly to the input array, or to `NULL`. (In either case the input array will also be used as the temporary working array.)

The minimum size of the twiddle table must be `n`/2. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set appropriately. If the size of the twiddle table is `x`, then `twiddle_stride` must be set to (2*`x`)/`n`.

If a larger twiddle table is being used, the twiddle stride must be adjusted to be equal to the `fft` size of the table generated divided by the fft size of the table being used.

The library function `twidfft` (on page 2-258) can be used to compute the required twiddle table. The coefficients generated are positive cosine coefficients for the real part and negative sine coefficients for the imaginary part.

For the ADSP-21xxx SIMD processors the library also contains the `cfftf` function (on page 2-73), which is an optimized implementation of a complex FFT using a fast radix-2 algorithm. The `cfftf` function however imposes certain memory alignment requirements that may not be appropriate for some applications.

The function returns the address of the `output` array.

For the ADSP-21xxx SIMD processors the `cfft` function uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

Algorithm

The following equation is the basis of the algorithm.

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

Error Conditions

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

Example

```
#include <filter.h>

#define N_FFT  64

complex_float  input[N_FFT];
complex_float  output[N_FFT];
complex_float  temp[N_FFT];
int            twiddle_stride = 1;

complex_float  pm twiddle[N_FFT/2];

   /* Populate twiddle table */
twidfft(twiddle, N_FFT);
   /* Compute Fast Fourier Transform */
cfft(input, temp, output, twiddle, twiddle_stride, N_FFT);
```

See Also

cfftf (SHARC SIMD Processors), cfftN (SHARC SIMD Processors), fft_magnitude, ifft, rfft, twidfft

The `cfft` function makes use of circular buffers. Refer to the "Interrupts and Circular Buffering" section of Chapter 1, "Compiler", in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* for information on interrupt dispatcher considerations when circular buffers are used within an application.

cfft_mag (SHARC SIMD Processors)

cfft magnitude

Synopsis

```
#include <filter.h>

float *cfft_mag (complex_float dm input[],
                 float dm output[],
                 int fftsize);
```

Description

The `cfft_mag` function computes a normalized power spectrum from the output signal generated by a `cfft` or `cfftN` function. The size of the signal and the size of the power spectrum is `fftsize`.

The function returns a pointer to the `output` matrix.

The Nyquist frequency is located at `(fftsize/2) + 1`.

Algorithm

The algorithm used to calculate the normalized power spectrum is:

$$magnitude(z) = \frac{\sqrt{Re(a_z)^2 + Im(a_z)^2}}{fftsize}$$

where:

z = {0, 1, ..., `fftsize`-1}
a is the input vector `input`

Error Conditions

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

Example

```
#include <filter.h>
#define N 64

complex_float fft_input[N];
complex_float fft_output[N];
float spectrum[N];

cfft64 (fft_input, fft_output);
cfft_mag (fft_output, spectrum, N);
```

See Also

cfft, cfftN (SHARC SIMD Processors), fft_magnitude, fftf_magnitude (SHARC SIMD Processors), rfft_mag (SHARC SIMD Processors)

By default, this function uses SIMD. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

cfftN

N-point complex radix-2 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 `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. 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. 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` are two floating-point arrays of N points. The array `real_input` contains the real components of the complex signal, and the array `imag_input` contains the imaginary components.

If there are fewer than N actual data points, you must pad the arrays 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.

If the input data can be overwritten, then the `cfftN` functions allow the array `real_input` to share the same memory as the array `real_output`, and `imag_input` to share the same memory as `imag_output`. This improves memory usage, but at the cost of run-time performance.

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

> The `cfftN` library functions have not been optimized for SHARC SIMD processors. Instead, applications that run on SHARC SIMD processors should use the FFT functions that are defined in the header file `filter.h`, and described under “cfftN (SHARC SIMD Processors)” on page 2-70.

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];

cfft2048 (real_input, imag_input, real_output, imag_output);
```

See Also

cfft, cfftN (SHARC SIMD Processors), fft_magnitude, ifftN, rfftN

The `cfftN` functions make use of circular buffers. Refer to the "Interrupts and Circular Buffering" section of Chapter 1, "Compiler", in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* for information on interrupt dispatcher considerations when circular buffers are used within an application.

cfftN (SHARC SIMD Processors)

N-point complex input FFT

Synopsis

```
#include <filter.h>

complex_float *cfft65536 (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft32768 (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft16384 (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft8192  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft4096  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft2048  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft1024  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft512   (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft256   (complex_float dm input[],
                          complex_float dm output[]);
```

```
complex_float *cfft128   (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft64    (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft32    (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft16    (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft8     (complex_float dm input[],
                          complex_float dm output[]);
```

Description

These `cfftN` functions are defined in the header file `filter.h`. They have been optimized to take advantage of the SIMD capabilities of the ADSP-211xx, ADSP-212xx, ADSP-213xx, and ADSP-214xx processors. Therefore, they are not supported by the ADSP-210xx processor family. These FFT functions require complex arguments to ensure that the real and imaginary parts are interleaved in memory and thus are accessible in a single cycle using the wider data bus of the processor.

Each of these `cfftN` functions computes the N-point radix-2 Fast Fourier Transform (CFFT) of its complex 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. 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. Call a particular function by substituting the number of points for N, as in `cfft8 (input, output);`

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. Optimal memory usage can be achieved by specifying the input array as the output array, but at the cost of run-time performance.

The `cfftN()` function returns a pointer to the `output` array.

The `cfftN` functions use the input array as an intermediate workspace. If the input data is to be preserved it must first be copied to a safe location before calling these functions.

Error Conditions

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

Example

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

complex_float input[N], output[N];

cfft2048 (input, output);
```

See Also

cfft, cfftf (SHARC SIMD Processors), fft_magnitude, ifftN, rfftN

By default these functions use SIMD. For more information, refer to “Implications of Using SIMD Mode”.

cfftf (SHARC SIMD Processors)

Fast N-point complex radix-2 Fast Fourier Transform

Synopsis

```
#include <filter.h>

void cfftf (float data_real[], float data_imag[],
            float temp_real[], float temp_imag[],
            const float twid_real[],
            const float twid_imag[],
            int n);
```

Description

The `cfftf` function transforms the time domain complex input signal sequence to the frequency domain by using the accelerated version of the Discrete Fourier Transform known as a Fast Fourier Transform or FFT. It decimates in frequency using an optimized radix-2 algorithm.

The array `data_real` contains the real part of a complex input signal, and the array `data_imag` contains the imaginary part of the signal. On output, the function overwrites the data in these arrays and stores the real part of the FFT in `data_real`, and the imaginary part of the FFT in `data_imag`. If the input data is to be preserved, it must first be copied to a safe location before calling this function. The argument `n` represents the number of points in the FFT; it must be a power of 2 and must be at least 64.

The `cfftf` function has been designed for optimal performance and requires that the arrays `data_real` and `data_imag` are aligned on an address boundary that is a multiple of the FFT size. For certain applications, this alignment constraint may not be appropriate; in such cases, the application should call the `cfft` function instead with no loss of facility (apart from performance).

The arrays `temp_real` and `temp_imag` are used as intermediate temporary buffers and should each be of size `n`.

The twiddle table is passed in using the arrays `twid_real` and `twid_imag`. The array `twid_real` contains the positive cosine factors, and the array `twid_imag` contains the negative sine factors; each array should be of size `n/2` . The `twidfftf` function (on page 2-261) may be used to initialize the twiddle table arrays.

It is recommended that the arrays containing real parts (`data_real`, `temp_real`, and `twid_real`) are allocated in separate memory blocks from the arrays containing imaginary parts (`data_imag`, `temp_imag`, and `twid_imag`); otherwise, the performance of the function degrades.

Error Conditions

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

Example

```
#include <filter.h>

#define FFT_SIZE 1024

#pragma align 1024
float dm input_r[FFT_SIZE];
#pragma align 1024
float pm input_i[FFT_SIZE];

float dm temp_r[FFT_SIZE];
float pm temp_i[FFT_SIZE];
float dm twid_r[FFT_SIZE/2];
float pm twid_i[FFT_SIZE/2];

twidfftf(twid_r,twid_i,FFT_SIZE);
cfftf(input_r,input_i,
```

```
temp_r,temp_i,
twid_r,twid_i,FFT_SIZE);
```

See Also

cfft, cfftN (SHARC SIMD Processors), fftf_magnitude (SHARC SIMD Processors), ifftf (SHARC SIMD Processors), rfftf_2 (SHARC SIMD Processors), twidfftf (SHARC SIMD Processors)

The `cfftf` function has been implemented to make highly efficient use of the processor's SIMD capabilities and long word addressing mode. The function therefore imposes the following restrictions:

- All the arrays that are passed to the function must be allocated in internal memory. The DSP run-time library does not contain a version of the function that can be used with data in external memory.
- The function should not be used with any application that relies on the `-reserve register[, register...]` switch.

For more information, refer to "Implications of Using SIMD Mode" and "Using Data in External Memory".

circindex

Perform circular buffer operation on loop index

Synopsis

```
#include <processor_include.h>
int circindex(ptrdiff_t index, ptrdiff_t incr, size_t num_items);
```

Description

The `circindex` function is used within a loop in order to implement a circular buffer operation in C/C++. When optimization is enabled, the operation is implemented using the appropriate hardware features (B registers and L registers) of the SHARC architecture. The `circindex` function is used to increment or decrement an index in a loop and this index should be used to access memory locations.

The argument `index` represents the index variable, `incr` represents the value by which the index should be incremented on each iteration, and `num_items` represents the size of the circular buffer.

Error Conditions

The `circindex` function does not return an error code.

Example

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

int x[10] = {1,2,3,4,5,6,7,8,9,10};
int y[10] = {2,3,4,5,6,7,8,9,10,11};
```

```
int dot (const int *a, const int *b)
{
   int i, ci = 0;
   long s = 0;

   /* This will calculate the product for the first 5 elements
    * in each array only. As the loop count is 10, each sum will
    * be calculated twice.
    * Note that each array is indexed using 'ci'. */

   for (i = 0; i < 10; i++) {
      s += a[ci] * b[ci];
      ci = circindex(ci, 1, 5);      // Increment the index
   }

   return s;
}

void
main()
{
   int result;
   result = dot(x,y);
   printf("Result is %d\n", result);      // Result is 140
}
```

See Also

circptr

circptr

Perform circular buffer operation on a pointer

Synopsis

```
#include <processor_include.h>
void* circptr(const void *ptr, ptrdiff_t incr,
              const void *base, size_t buflen);
```

Description

The `circptr` function is used within a loop in order to implement a circular buffer operation in C/C++. When optimization is enabled, the operation is implemented using the appropriate hardware features (B registers and L registers) of the SHARC processor architecture. The `circptr` function is used to increment or decrement a pointer variable in a loop.

When used with a `PM` qualified circular buffer, the result of the circular buffer function should be cast to `(void pm *)`.

The argument `ptr` represents the pointer that is being used for the circular buffer, `incr` represents the value by which the circular buffer should be incremented, `base` represents the array on which the circular buffer operates, and `buflen` represents the size of the circular buffer.

Error Conditions

The `circptr` function does not return an error code.

Example

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

int    x[10] = {1,2,3,4,5,6,7,8,9,10};
int pm y[10] = {2,3,4,5,6,7,8,9,10,11};
```

```
int dot (const int *a, const int pm *b)
{
   int i;
   long s = 0;
   const int *cba;
   const int pm *cbb;

   /* This will calculate the product for the first 5 elements
      in each array only. As the loop count is 10,each sum will
      be calculated twice. */

   cba = a;
   cbb = b;
   for (i = 0; i < 10; i++) {
      s += *cba * *cbb;
      cba = circptr(cba, 1, a, 5);              // Increment cba
      cbb = (void pm *)circptr(cbb, 1, b, 5);   // Increment cbb
   }

   return s;
}

void
main()
{
   int result;
   result = dot(x,y);
   printf("Result is %d\n", result);             // Result is 140
}
```

See Also

circindex

cmatmadd

Complex matrix + matrix addition

Synopsis

```
#include <cmatrix.h>

complex_float *cmatmaddf (complex_float dm *output,
                          const complex_float dm *a,
                          const complex_float dm *b,
                          int rows, int cols);

complex_double *cmatmadd (complex_double dm *output,
                          const complex_double dm *a,
                          const complex_double dm *b,
                          int rows, int cols);

complex_long_double *cmatmaddd (complex_long_double dm *output,
                          const complex_long_double dm *a,
                          const complex_long_double dm *b,
                          int rows, int cols);
```

Description

The `cmatmadd` functions perform a complex matrix addition of the input matrix `a[][]` with input complex matrix `b[][]`, and store the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]`, `b[rows][cols]`, and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The `cmatmadd` functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS 4
#define COLS 8

complex_double a[ROWS][COLS], *a_p = (complex_double *) (&a);
complex_double b[ROWS][COLS], *b_p = (complex_double *) (&b);
complex_double c[ROWS][COLS], *res_p = (complex_double *) (&c);

cmatmadd (res_p, a_p, b_p, ROWS, COLS);
```

See Also

cmatmmlt, cmatmsub, cmatsadd, matmadd

For the ADSP-21xxx SIMD processors (and `cmatmadd`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

cmatmmlt

Complex matrix * matrix multiplication

Synopsis

```
#include <cmatrix.h>

complex_float *cmatmmltf (complex_float dm *output,
                          const complex_float dm *a,
                          const complex_float dm *b,
                          int a_rows, int a_cols, int b_cols);

complex_double *cmatmmlt (complex_double dm *output,
                          const complex_double dm *a,
                          const complex_double dm *b,
                          int a_rows, int a_cols, int b_cols);

complex_long_double *cmatmmltd (complex_long_double dm *output,
                          const complex_long_double dm *a,
                          const complex_long_double dm *b,
                          int a_rows, int a_cols, int b_cols);
```

Description

The `cmatmmlt` functions perform a complex matrix multiplication of the input matrices `a[][]` and `b[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[a_rows][a_cols]`, `b[a_cols][b_cols]`, and `output[a_rows][b_cols]`. The functions return a pointer to the output matrix.

Algorithm

Complex matrix multiplication is defined by the following algorithm:

$$Re(c_{i,j}) = \sum_{l=0}^{a\_cols-1} (Re(a_{i,l})) \bullet (Re(b_{l,j})) - Im(a_{i,l}) \bullet Im(b_{l,j})$$

$$Im(c_{i,j}) = \sum_{l=0}^{a\_cols-1} (Re(a_{i,l})) \bullet (Im(b_{l,j})) + Im(a_{i,l}) \bullet Re(b_{l,j})$$

where

i = {0, 1, 2, ..., `a_rows`-1}, j = {0, 1, 2, ..., `b_cols`-1}

Error Conditions

The `cmatmmlt` functions do not return an error condition.

Example

```
#include <cmatrix.h>

#define ROWS_1 4
#define COLS_1 8
#define COLS_2 2

complex_double a[ROWS_1][COLS_1], *a_p = (complex_double *) (&a);
complex_double b[COLS_1][COLS_2], *b_p = (complex_double *) (&b);
complex_double c[ROWS_1][COLS_2], *r_p = (complex_double *) (&c);

cmatmmlt (r_p, a_p, b_p, ROWS_1, COLS_1, COLS_2);
```

See Also

cmatmadd, cmatmsub, cmatsmlt, matmmlt

The `cmatmmlt` functions make use of circular buffers. Refer to the "Interrupts and Circular Buffering" section of Chapter 1, "Compiler", in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* for information on interrupt dispatcher considerations when circular buffers are used within an application.

cmatmsub

Complex matrix – matrix subtraction

Synopsis

```
#include <cmatrix.h>

complex_float *cmatmsubf (complex_float dm *output,
                         const complex_float dm *a,
                         const complex_float dm *b,
                         int rows, int cols);

complex_double *cmatmsub (complex_double dm *output,
                         const complex_double dm *a,
                         const complex_double dm *b,
                         int rows, int cols);

complex_long_double *cmatmsubd (complex_long_double dm *output,
                         const complex_long_double dm *a,
                         const complex_long_double dm *b,
                         int rows, int cols);
```

Description

The `cmatmsub` functions perform a complex matrix subtraction between the input matrices `a[][]` and `b[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]`, `b[rows][cols]`, and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The `cmatmsub` functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS 4
#define COLS 8

complex_double a[ROWS][COLS], *a_p = (complex_double *) (&a);
complex_double b[ROWS][COLS], *b_p = (complex_double *) (&b);
complex_double c[ROWS][COLS], *res_p = (complex_double *) (&c);

cmatmsub (res_p, a_p, b_p, ROWS, COLS);
```

See Also

cmatmadd, cmatmmlt, cmatssub, matmsub

For the ADSP-21xxx SIMD processors the `cmatmsubf` function (and `cmatmsub`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

cmatsadd

Complex matrix + scalar addition

Synopsis

```
#include <cmatrix.h>

complex_float *cmatsaddf (complex_float dm *output,
                         const complex_float dm *a,
                         complex_float scalar,
                         int rows, int cols);

complex_double *cmatsadd (complex_double dm *output,
                         const complex_double dm *a,
                         complex_double scalar,
                         int rows, int cols);

complex_long_double *cmatsaddd (complex_long_double dm *output,
                         const complex_long_double dm *a,
                         complex_long_double scalar,
                         int rows, int cols);
```

Description

The `cmatsadd` functions add a complex scalar to each element of the complex input matrix `a[][]` and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The `cmatsadd` functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS 4
#define COLS 8

complex_double a[ROWS][COLS], *a_p = (complex_double *) (&a);
complex_double c[ROWS][COLS], *res_p = (complex_double *) (&c);
complex_double z;

cmatsadd (res_p, a_p, z, ROWS, COLS);
```

See Also

cmatsmlt, cmatssub, cmatmadd, matsadd

For the ADSP-21xxx SIMD processors the `cmatsaddf` function (and `cmatsadd`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

cmatsmlt

Complex matrix * scalar multiplication

Synopsis

```
#include <cmatrix.h>

complex_float *cmatsmltf (complex_float dm *output,
                          const complex_float dm *a,
                          complex_float scalar
                          int rows, int cols);

complex_double *cmatsmlt (complex_double dm *output,
                          const complex_double dm *a,
                          complex_double scalar,
                          int rows, int cols);

complex_long_double *cmatsmltd (complex_long_double dm *output,
                          const complex_long_double dm *a,
                          complex_long_double scalar,
                          int rows, int cols);
```

Description

The `cmatsmlt` functions multiply each element of the complex input matrix `a[][]` with a complex scalar, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`. The functions return a pointer to the output matrix.

Algorithm

Complex matrix multiplication is defined by the following algorithm:

$$Re(c_{i,j}) = \sum_{k=0}^{a\_cols-1} (Re(a_{i,k})) \bullet (Re(b_{k,j})) - Im(a_{i,k}) \bullet Im(b_{k,j})$$

$$Im(c_{i,j}) = \sum_{k=0}^{a\_cols-1} (Re(a_{i,k})) \bullet (Im(b_{k,j})) + Im(a_{i,k}) \bullet Re(b_{k,j})$$

where

i = {0, 1, 2, ..., `rows`-1}, j = {0, 1, 2, ..., `cols`-1}

Error Conditions

The `cmatsmlt` functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS 4
#define COLS 8

complex_double a[ROWS][COLS], *a_p = (complex_double *) (&a);
complex_double c[ROWS][COLS], *res_p = (complex_double *) (&c);
complex_double z;

cmatsmlt (res_p, a_p, z, ROWS, COLS);
```

See Also

cmatsadd, cmatssub, cmatmmlt, matsmlt

cmatssub

Complex matrix – scalar subtraction

Synopsis

```
#include <cmatrix.h>

complex_float *cmatssubf (complex_float dm *output,
                         const complex_float dm *a,
                         complex_float scalar,
                         int rows, int cols);

complex_double *cmatssub (complex_double dm *output,
                         const complex_double dm *a,
                         complex_double scalar,
                         int rows, int cols);

complex_long_double *cmatssubd (complex_long_double dm *output,
                         const complex_long_double dm *a,
                         complex_long_double scalar,
                         int rows, int cols);
```

Description

The `cmatssub` functions subtract a complex scalar from each element of the complex input matrix `a[][]` and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The `cmatssub` functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS 4
#define COLS 8

complex_double a[ROWS][COLS], *a_p = (complex_double *) (&a);
complex_double c[ROWS][COLS], *res_p = (complex_double *) (&c);
complex_double z;

cmatssub (res_p, a_p, z, ROWS, COLS);
```

See Also

cmatsadd, cmatsmlt, cmatmsub, matssub

For the ADSP-21xxx SIMD processors the `cmatssubf` function (and `cmatssub`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

cmlt

Complex multiplication

Synopsis

```
#include <complex.h>

complex_float cmltf (complex_float a, complex_float b);
complex_double cmlt (complex_double a, complex_double b);
complex_long_double cmltd (complex_long_double a,
                           complex_long_double b);
```

Description

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

Error Conditions

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

Example

```
#include <complex.h>

complex_float x = {3.0,11.0};
complex_float y = {1.0, 2.0};
complex_float z;

z = cmltf(x,y);    /* z.re = -19.0, z.im = 17.0 */
```

See Also

cadd, cdiv, csub

conj

Complex conjugate

Synopsis

```
#include <complex.h>

complex_float conjf (complex_float a);
complex_double conj (complex_double a);
complex_long_double conjd (complex_long_double a);
```

Description

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

Error Conditions

The complex conjugate functions do not return any error conditions.

Example

```
#include <complex.h>

complex_double x = {2.0,8.0};
complex_double z;

z = conj(x);      /* z = (2.0,-8.0) */
```

See Also

No related function.

convolve

Convolution

Synopsis

```
#include <filter.h>

float *convolve (const float a[], int asize,
                 const float b[], int bsize, float output[]);
```

Description

The convolution function calculates the convolution of the input vectors `a[]` and `b[]` , and returns the result in the vector `output[]`. The lengths of these vectors are `a[asize]`, `b[bsize]`, and `output[asize+bsize-1]`.

The `convolve` function returns a pointer to the output vector.

Algorithm

Convolution of two vectors is defined as:

$$c_k = \sum_{j=m}^{n} a_j \bullet b_{(k-j)}$$

where

k = {0, 1, ..., `asize+bsize`-2}
m = max(0, `k+1-bsize`)
n = min(k, `asize`-1)

Error Conditions

The convolution function does not return an error condition.

Example

```
#include <filter.h>

float input[81];
float response[31];
float output[81 + 31 -1];

convolve(input,81,response,31,output);
```

See Also

crosscorr

The `convolve` function makes use of circular buffers. Refer to the "Interrupts and Circular Buffering" section of Chapter 1, "Compiler", in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* for information on interrupt dispatcher considerations when circular buffers are used within an application.

copysign

Copy the sign of the floating-point operand.

Synopsis

```
#include <math.h>

float copysignf (float x, float y);
double copysign (double x, double y);
long double copysignd (long double x, long double y);
```

Description

The `copysign` functions copy the sign of the second argument `y` to the first argument `x` without changing its exponent or mantissa.

The `copysignf` function is a built-in function which is implemented with an `Fn=Fx COPYSIGN Fy` instruction. The `copysign` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

The `copysign` functions do 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 related function.

cot

Cotangent

Synopsis

```
#include <math.h>

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

Description

The cotangent functions return the cotangent of their argument. The input is interpreted as radians.

Error Conditions

The input argument `x` for `cotf` must be in the domain [-1.647e6, 1.647e6] and the input argument for `cotd` must be in the domain [-4.21657e8, 4.21657e8]. The functions return zero if `x` is outside their domain.

Example

```
#include <math.h>
#define PI  3.141592653589793

double d;
float r;

d = cot (-PI/4.0);    /* d = -1.0 */
r = cotf( PI/4.0F);   /* r = 1.0  */
```

See Also

tan

crosscoh

Cross-coherence

Synopsis

```
#include <stats.h>

float *crosscohf (float dm out[],
                  const float dm x[], const float dm y[],
                  int samples, int lags);

double *crosscoh (double dm out[],
                  const double dm x[], const double dm y[],
                  int samples, int lags);

long double *crosscohd (long double dm out[],
                        const long double dm x[],
                        const long double dm y[],
                        int samples, int lags);
```

Description

The cross-coherence functions compute 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`. The functions return a pointer to the output array `out[]` of length `lags`.

Algorithm

The following equation is the basis of the algorithm.

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

where:

k = {0, 1, ..., `lags`-1}
a = `x`
b = `y`
c = `coherence`
$\bar{a}$ is the mean value of input vector `a`
$\bar{b}$ is the mean value of input vector `b`

Error Conditions

The cross-coherence functions do not return an error condition.

Example

```
#include <stats.h>

#define SAMPLES 1024
#define LAGS      16

double excitation[SAMPLES], y[SAMPLES];
double response[LAGS];
int lags = LAGS;

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

See Also

autocoh, autocorr, crosscorr

> For the ADSP-21xxx SIMD processors the `crosscohf` function (and `crosscoh`, if `doubles` are the same size as `floats`) uses SIMD; refer to “Implications of Using SIMD Mode” on page 2-23 for more information.

crosscorr

Cross-correlation

Synopsis

```
#include <stats.h>

float *crosscorrf (float dm out[],
                   const float dm x[], const float dm y[],
                   int samples, int lags);

double *crosscorr (double dm out[],
                   const double dm x[], const double dm y[],
                   int samples, int lags);

long double *crosscorrd (long double dm out[],
                         const long double dm x[],
                         const long double dm y[],
                         int samples, int lags);
```

Description

The cross-correlation functions perform a cross-correlation between two signals. The cross-correlation is the sum of the scalar products of the signals in which the signals are displaced in time with respect to one another. The signals to be correlated are given by the input arrays `x[]` and `y[]`. The length of the input arrays is given by `samples`. The functions return a pointer to the output data array `out[]` of length `lags`.

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

Algorithm

The following equation is the basis of the algorithm.

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

where:

k = {0, 1, ..., `lags`-1}
a = `x`
b = `y`
n = `samples`

Error Conditions

The cross-correlation functions do not return an error condition.

Example

```
#include <stats.h>
#define SAMPLES 1024
#define LAGS      16

double excitation[SAMPLES], y[SAMPLES];
double response[LAGS];
int lags = LAGS;

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

See Also

autocoh, autocorr, crosscoh

For the ADSP-21xxx SIMD processors the `crosscorrf` function (and `crosscorr`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

csub

Complex subtraction

Synopsis

```
#include <complex.h>

complex_float csubf (complex_float a, complex_float b);
complex_double csub (complex_double a, complex_double b);
complex_long_double csubd (complex_long_double a,
                           complex_long_double b);
```

Description

The `csub` functions subtract the two complex values `a` and `b`, and return the result.

Error Conditions

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

Example

```
#include <complex.h>

complex_float x = {9.0,16.0};
complex_float y = {1.0,-1.0};
complex_float z;

z = csubf(x,y);      /* z.re = 8.0, z.im = 17.0 */
```

See Also

cadd, cdiv, cmlt

cvecdot

Complex vector dot product

Synopsis

```
#include <cvector.h>

complex_float cvecdotf (const complex_float dm a[],
                        const complex_float dm b[], int samples);

complex_double cvecdot (const complex_double dm a[],
                        const complex_double dm b[], int samples);

complex_long_double cvecdotd (const complex_long_double dm a[],
                              const complex_long_double dm b[],
                              int samples);
```

Description

The `cvecdot` functions compute the complex dot product of the complex vectors `a[]` and `b[]`, which are `samples` in size. The scalar result is returned by the function.

Algorithm

The algorithm for a complex dot product is given by:

$$Im(c_i) = \sum_{l=0}^{n-1} (Re(a_i) \bullet (Im(b_i)) + Im(a_i) \bullet Re(b_i))$$

where:

$$Re(c_i) = \sum_{l=0}^{n-1} (Re(a_i) \bullet (Re(b_i)) - Im(a_i) \bullet Im(b_i))$$

i = {0, 1 ,2, ..., `samples`-1}

Error Conditions

The `cvecdot` functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float x[N], y[N];
complex_float answer;

answer = cvecdotf (x, y, N);
```

See Also

vecdot

cvecsadd

Complex vector + scalar addition

Synopsis

```
#include <cvector.h>

complex_float *cvecsaddf (const complex_float dm a[],
                          complex_float scalar,
                          complex_float dm output[], int samples);

complex_double *cvecsadd (const complex_double dm a[],
                          complex_double scalar,
                          complex_double dm output[], int
                          samples);

complex_long_double *cvecsaddd (const complex_long_double dm a[],
                                complex_long_double scalar,
                                complex_long_double dm output[],
                                int samples);
```

Description

The `cvecsadd` functions compute the sum of each element of the complex vector `a[]`, added to the complex scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The `cvecsadd` functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input[N], result[N];
complex_float x;

cvecsaddf (input, x, result, N);
```

See Also

cvecsmlt, cvecssub, cvecvadd, vecsadd

For the ADSP-21xxx SIMD processors the `cvecsaddf` function (and `cvecsadd`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

cvecsmlt

Complex vector * scalar multiplication

Synopsis

```
#include <cvector.h>

complex_float *cvecsmltf (const complex_float dm a[],
                          complex_float scalar,
                          complex_float dm output[], int samples);

complex_double *cvecsmlt (const complex_double dm a[],
                          complex_double scalar,
                          complex_double dm output[], int
                          samples);

complex_long_double *cvecsmltd (const complex_long_double dm a[],
                                complex_long_double scalar,
                                complex_long_double dm output[],
                                int samples);
```

Description

The `cvecsmlt` functions compute the product of each element of the complex vector `a[]`, multiplied by the complex scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Complex vector by scalar multiplication is given by the formula:

$\mathrm{Re}(c_i) = \mathrm{Re}(a_i)*\mathrm{Re}(scalar) - \mathrm{Im}(a_i)*\mathrm{Im}(\mathrm{scalar})$

$\mathrm{Im}(c_i) = \mathrm{Re}(a_i)*\mathrm{Im}(\mathrm{scalar}) + \mathrm{Im}(a_i)*\mathrm{Re}(\mathrm{scalar})$

where: `i={0,1,2,...,samples-1}`

Error Conditions

The `cvecsmlt` functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input[N], result[N];
complex_float x;

cvecsmltf (input, x, result, N);
```

See Also

cvecsadd, cvecssub, cvecvmlt, vecsmlt

cvecssub

Complex vector – scalar subtraction

Synopsis

```
#include <cvector.h>

complex_float *cvecssubf (const complex_float dm a[],
                          complex_float scalar,
                          complex_float dm output[], int samples);

complex_double *cvecssub (const complex_double dm a[],
                          complex_double scalar,
                          complex_double dm output[], int
                          samples);

complex_long_double *cvecssubd (const complex_long_double dm a[],
                                complex_long_double scalar,
                                complex_long_double dm output[],
                                int samples);
```

Description

The `cvecssub` functions compute the difference of each element of the complex vector `a[]`, minus the complex scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The `cvecssub` functions do not return an error condition.

Example

```
#include <cvector.h>

#define N 100

complex_float input[N], result[N];
complex_float x;

cvecssubf (input, x, result, N);
```

See Also

cvecsadd, cvecsmlt, cvecvsub, vecssub

For the ADSP-21xxx SIMD processors the `cvecssubf` function (and `cvecssub`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

cvecvadd

Complex vector + vector addition

Synopsis

```
#include <cvector.h>

complex_float *cvecvaddf (const complex_float dm a[],
                          const complex_float dm b[],
                          complex_float dm output[], int samples);

complex_double *cvecvadd (const complex_double dm a[],
                          const complex_double dm b[],
                          complex_double dm output[], int
                          samples);

complex_long_double *cvecvaddd (const complex_long_double dm a[],
                                const complex_long_double dm b[],
                                complex_long_double dm output[],
                                int samples);
```

Description

The `cvecvadd` functions compute the sum of each of the elements of the complex vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The `cvecvadd` functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input_1[N];
complex_float input_2[N], result[N];

cvecvaddf (input_1, input_2, result, N);
```

See Also

cvecsadd, cvecvmlt, cvecvsub, vecvadd

For the ADSP-21xxx SIMD processors the `cvecvaddf` function (and `cvecvadd`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to “Implications of Using SIMD Mode” on page 2-23 for more information.

cvecvmlt

Complex vector * vector multiply

Synopsis

```
#include <cvector.h>

complex_float *cvecvmltf (const complex_float dm a[],
                          const complex_float dm b[],
                          complex_float dm output[], int samples);

complex_double *cvecvmlt (const complex_double dm a[],
                          const complex_double dm b[],
                          complex_double dm output[], int
                          samples);

complex_long_double *cvecvmltd (const complex_long_double dm a[],
                                const complex_long_double dm b[],
                                complex_long_double dm output[],
                                int samples);
```

Description

The `cvecvmlt` functions compute the product of each of the elements of the complex vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Complex vector multiplication is given by the formula:

$$\mathrm{Re}(c_i) = \mathrm{Re}(a_i)*\mathrm{Re}(b_i) - \mathrm{Im}(a_i)*\mathrm{Im}(b_i)$$

$$\mathrm{Im}(c_i) = \mathrm{Re}(a_i)*\mathrm{Im}(b_i) + \mathrm{Im}(a_i)*\mathrm{Re}(b_i)$$

where: i = {0, 1, 2, ..., `samples`-1}

Error Conditions

The `cvecvmlt` functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input_1[N];
complex_float input_2[N], result[N];

cvecvmltf (input_1, input_2, result, N);
```

See Also

cvecsmlt, cvecvadd, cvecvsub, vecvmlt

For the ADSP-21xxx SIMD processors restrictions apply to this function if the data is placed in external memory. See "Using Data in External Memory" on page 2-24 for more information.

cvecvsub

Complex vector – vector subtraction

Synopsis

```
#include <cvector.h>

complex_float *cvecvsubf (const complex_float dm a[],
                          const complex_float dm b[],
                          complex_float dm output[], int samples);

complex_double *cvecvsub (const complex_double dm a[],
                          const complex_double dm b[],
                          complex_double dm output[], int
                          samples);

complex_long_double *cvecvsubd (const complex_long_double dm a[],
                                const complex_long_double dm b[],
                                complex_long_double dm output[],
                                int samples);
```

Description

The `cvecvsub` functions compute the difference of each of the elements of the complex vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The `cvecvsub` functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input_1[N];
complex_float input_2[N], result[N];

cvecvsubf (input_1, input_2, result, N);
```

See Also

cvecssub, cvecvadd, cvecvmlt,vecvsub

For the ADSP-21xxx SIMD processors the `cvecvsubf` function (and `cvecvsub`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

dma_disable

Clears the channel's DMA enable bit

Synopsis

```
#include <dma.h>
static int dma_disable (int dma_channel);
```

Description

The `dma_disable` function clears the channel's DMA enable (DEN) bit.

Error Conditions

If the channel is invalid, `dma_status` returns -1.

See Also

dma_status, lavg

dma_enable

Sets the channel's DMA enable bit

Synopsis

```
#include <dma.h>
static int dma_enable (int dma_channel);
```

Description

The `dma_enable` function sets the channel's DMA enable (DEN) bit.

Error Conditions

If the channel is invalid, `dma_status` returns -1.

See Also

dma_disable, dma_setup, dma_status

dma_setup

Sets up the DMA channel

Synopsis

```
#include <dma.h>
static int dma_setup(int dma_channel, struct __dma_control_word
dma_control_word);
```

Description

The `dma_setup` function sets up the DMA channel with the values in the DMA control word.

Error Conditions

If the channel is invalid, `dma_status` returns -1.

See Also

dma_enable, dma_disable, dma_status

dma_status

Returns the status of the DMA channel

Synopsis

```
#include <dma.h>
static int dma_status (int dma_channel);
```

Description

The `dma_status` function returns the status of the DMA channel.

Error Conditions

If the channel is invalid, `dma_status` returns -1.

See Also

dma_enable, dma_disable, dma_setup

favg

Mean of two values

Synopsis

```
#include <math.h>

float favgf (float x, float y);
double favg (double x, double y);
long double favgd (long double x, long double y);
```

Description

The `favg` functions return the mean of their two arguments.

The `favgf` function is a built-in function which is implemented with an `Fn=(Fx+Fy)/2` instruction. The `favg` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

The `favg` 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

fclip

Clip

Synopsis

```
#include <math.h>

float fclipf (float x, float y);
double fclip (double x, double y);
long double fclipd (long double x, long double y);
```

Description

The `fclip` functions return the first argument if its absolute value is less than the absolute value of the second argument, otherwise they return the absolute value of the second argument if the first is positive, or minus the absolute value if the first argument is negative.

The `fclipf` function is a built-in function which is implemented with an `Fn=CLIP Fx BY Fy` instruction. The `fclip` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

The `fclip` 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

fft_magnitude

FFT magnitude

Synopsis

```
#include <filter.h>

float *fft_magnitude (complex_float  input[],
                      float          output[],
                      int            fftsize,
                      int            mode);
```

Description

The `fft_magnitude` function computes a normalized power spectrum from the output signal generated by an FFT function; the `mode` parameter is used to specify which FFT function has been used to generate the input array.

If the input array has been generated by the `cfft` function, the mode must be set to `0`. In this case the input array and the power spectrum are of size `fftsize`.

If the input array has been generated by the `rfft` function, `mode` must be set to `2`. In this case the input array and the power spectrum are of size `((fftsize / 2) + 1)`.

For SHARC SIMD processors, the `fft_magnitude` function may also be used to calculate the power spectrum of an FFT that was generated by the `cfftN` and `rfftN` functions. If the input array has been generated by the `rfftN` function, then `mode` must be set to 1, and the size of the input array and the power spectrum will be `(fftsize / 2)`. If the input array was generated by the `cfftN` function, then the mode must be set to 0 and the size of the input array and the power spectrum will be `fftsize` (as for the `cfft` function above).

The `fft_magnitude` function returns a pointer to the output.

For the ADSP-21xxx SIMD processors the `fft_magnitude` function provides the same functionality as the `cfft_mag` and `rfft_mag` function does. In addition, it provides a real FFT power spectrum that includes the Nyquist frequency (only in conjunction with the `rfft` function).

For the ADSP-21xxx SIMD processors the `fft_magnitude` function uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

Error Conditions

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

Algorithm (ADSP-210xx Processor)

For mode 0 (`cfft` generated input):

$$magnitude(z) = \frac{\sqrt{Re(a_z)^2 + Im(a_z)^2}}{fftsize}$$

For mode 2 (`rfft` generated input):

$$magnitude(z) = 2 \times \frac{\sqrt{Re(a_z)^2 + Im(a_z)^2}}{fftsize}$$

Algorithm (ADSP-21xxx SIMD Processors)

For mode 0 (`cfft` and `cfftN` generated input):

$$magnitude(z) = \frac{\sqrt{Re(a_z)^2 + Im(a_z)^2}}{fftsize}$$

For mode 1 and 2 (`rfftN` and `rfft` generated input):

$$magnitude(z) = 2 \times \frac{\sqrt{Re(a_z)^2 + Im(a_z)^2}}{fftsize}$$

Example

```
#include <filter.h>

#define N_FFT 64
#define N_RFFT_OUT ((N_FFT / 2) + 1)

/* Data for real FFT */
float rfft_input[N_FFT];
complex_float rfft_output[N_RFFT_OUT];
complex_float rfftN_output[N_RFFT_OUT - 1];

/* Data for complex FFT */
complex_float cfft_input[N_FFT];
complex_float cfft_output[N_RFFT_OUT];

complex_float pm twiddle[N_FFT / 2];
complex_float temp[N_FFT];
float *tmp = (float*)temp;
```

```
/* Power Spectrums */
float rspectrum[N_RFFT_OUT];
float rNspectrum[N_RFFT_OUT - 1];
float cspectrum[N_FFT];

/* Initialize */
twidfft(twiddle, N_FFT);

/* Power spectrum using rfft */
rfft (rfft_input, tmp, rfft_output, twiddle, 1, N_FFT);
fft_magnitude (rfft_output, rspectrum, N_FFT, 2);

#if defined(__SIMDSHARC__)
   rfft64 (rfft_input, rfftN_output);
   fft_magnitude (rfftN_output, rNspectrum, N_FFT, 1);
#endif
/* Power spectrum using cfft */
cfft (cfft_input, temp, cfft_output, twiddle, 1, N_FFT);
fft_magnitude (cfft_output, cspectrum, N_FFT, 0);
```

See Also

cfft, cfftN (SHARC SIMD Processors), cfft_mag (SHARC SIMD Processors), fftf_magnitude (SHARC SIMD Processors), rfft, rfft_mag (SHARC SIMD Processors), rfftN (SHARC SIMD Processors)

fftf_magnitude (SHARC SIMD Processors)

fftf magnitude

Synopsis

```
#include <filter.h>

float *fftf_magnitude (float  input_real[],
                       float  input_imag[],
                       float  output[],
                       int    fftsize,
                       int    mode);
```

Description

The `fftf_magnitude` function computes a normalized power spectrum from the output signal generated by one of the accelerated FFT functions `cfftf` or `rfftf_2`.

The `mode` argument is used to specify which FFT function has been used.

If the input array has been generated by the `cfftf` function, `mode` must be set to `0`. In this case the input array and the power spectrum are of size `fftsize`.

If the input array has been generated by the `rfftf_2` function, `mode` must be set to 2. In this case the input array and the power spectrum are of size `((fftsize / 2) + 1)`.

The `fftf_magnitude` function returns a pointer to the output.

Algorithm

For mode 0 (`cfftf` generated input):

$$magnitude(z) = \frac{\sqrt{Re(z)^2 + Im(z)^2}}{fftsize}$$

For mode 2 (`rfftf_2` generated input):

$$magnitude(z) = 2 \times \frac{\sqrt{Re(z)^2 + Im(z)^2}}{fftsize}$$

Error Conditions

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

Example

```
#include <filter.h>
#define N_FFT 64
#define N_RFFT_OUT ((N_FFT / 2) + 1)

float pm twiddle_re[N_FFT/2];
float dm twiddle_im[N_FFT/2];

#pragma align 64
float dm rfft1_re[N_FFT];
float dm rfft1_im[N_FFT];

#pragma align 64
float pm rfft2_re[N_FFT];
float pm rfft2_im[N_FFT];
```

```
#pragma align 64
float dm data_re[N_FFT];
float pm data_im[N_FFT];

#pragma align 64
float dm temp_re[N_FFT];
float pm temp_im[N_FFT];

float rspectrum_1[N_RFFT_OUT];
float rspectrum_2[N_RFFT_OUT];
float cspectrum[N_FFT];

twidfftf(twiddle_re, twiddle_im, N_FFT);

rfftf_2(rfft1_re, rfft1_im,
        rfft2_re, rfft2_im, twiddle_re, twiddle_im, N_FFT);
fftf_magnitude(rfft1_re, rfft1_im, rspectrum_1, N_FFT, 2);
fftf_magnitude(rfft2_re, rfft2_im, rspectrum_2, N_FFT, 2);

cfftf(data_re, data_im,
      temp_re, temp_im, twiddle_re, twiddle_im, N_FFT);
fftf_magnitude(data_re, data_im, cspectrum, N_FFT, 0);
```

See Also

cfftf (SHARC SIMD Processors), rfftf_2 (SHARC SIMD Processors)

By default, this function uses SIMD. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

fir

Finite impulse response (FIR) filter

Synopsis (Scalar-Valued Version)

```
#include <filters.h>

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

Synopsis (Vector-Valued Version)

ADSP-2106x Non-SIMD Processors

```
#include <filter.h>

float *fir_vec (const float  dm input[],
                float        dm output[],
                const float  pm coeffs[],
                float        dm state[],
                int             samples,
                int             taps);
```

ADSP-21xxx SIMD Processors

```
#include <filter.h>

float *fir (const float  dm input[],
            float        dm output[],
            const float  pm coeffs[],
            float        dm state[],
            int             samples,
            int             taps);
```

Description

The `fir` functions implement a finite impulse response (FIR) filter that is structured as a sum of products. The characteristics of the filter (passband, stop band, and so on) are dependent on the coefficients and the number of taps supplied by the calling program.

The scalar version of the `fir` function produces the filtered response of its input data sample, which it returns as the result of the function.

The vector versions of the `fir` function generate the filtered response of the input data `input` and store the result in the output vector `output`. The number of input samples and the length of the output vector is specified by the argument `samples`.

The number of coefficients is specified by the parameter `taps` and the coefficients must be stored in reverse order in the array `coeffs`; so `coeffs[0]` contains the last filter coefficient and `coeffs[taps-1]` contains the first coefficient. The array must be located in program memory data space so that the single-cycle dual-memory fetch of the processor can be used.

Each filter should have its own delay line, which is represented by the array `state`. The array contains a pointer into the delay line as its first element, followed by the delay line values. The length of the `state` array is therefore one greater than the number of taps.

The `state` array should be initially cleared to zero before calling the function for the first time, and should not otherwise be modified by the user program.

The library function uses the architecture's dual-data move instruction to provide simultaneous access to the filter coefficients (in PM data memory) and the delay line. When running on an ADSP-21367, ADSP-21368, ADSP-21369, ADSP-21371, or ADSP-21375 processor, the filter coefficients and the delay line must not both be allocated in external memory; otherwise, the function can generate an incorrect set of results. This occurs because in a dual-data move instruction, the hardware does not support both memory accesses allocated to external memory. Therefore, ensure that the filter coefficients or the delay line (or, optionally, both) are allocated in internal memory when running on one of the ADSP-213xx processors specified above.

The vector version of the `fir` functions return a pointer to the output vector; the scalar version of the function returns the filtered response of its input sample.

Error Conditions

The `fir` functions do not return an error condition.

Example

Scalar-Valued

```
#include <filters.h>

#define TAPS 10

float y;
float pm coeffs[TAPS];     /* coeffs array must be         */
                           /* initialized and in PM memory */
float state[TAPS+1];
int i;

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

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

Vector-Valued

```
#include <filter.h>

#define TAPS 10
#define SAMPLES 256

float input[SAMPLES];
float output[SAMPLES];
float pm coeffs[TAPS];      /* coeffs array must be         */
                            /* initialized and in PM memory */
float state[TAPS+1];
int i;

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

#if defined(__SIMDSHARC__)
     fir (input, output, coeffs, state, SAMPLES, TAPS);
#else
    fir_vec (input, output, coeffs, state, SAMPLES, TAPS);
#endif
```

See Also

biquad, fir_decima, fir_interp, iir

By default, the `fir` function for SHARC SIMD processors uses SIMD. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

The `fir_vec` function makes use of circular buffers. Refer to the "Interrupts and Circular Buffering" section of Chapter 1, "Compiler" in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* for information on interrupt dispatcher considerations when circular buffers are used within an application.

fir_decima

FIR-based decimation filter

Synopsis

```
#include <filter.h>

float *fir_decima (const float     input[],
                   float           output[],
                   const float pm  coefficients[],
                   float           delay[],
                   int             num_output_samples,
                   int             num_coeffs,
                   int             decimation_index);
```

Description

The `fir_decima` function implements a finite impulse response (FIR) filter defined by the coefficients and the delay line that are supplied in the call of `fir_decima`. The function produces the filtered response of its input data and then decimates.

The size of the output vector `output` is specified by the argument `num_output_samples`, which specifies the number of output samples to be generated. The input vector `input` should contain `decimation_index * num_output_samples` samples, where `decimation_index` represents the decimation index.

The characteristics of the filter are dependent on the number of coefficients and their values, and the decimation index supplied by the calling program.

The array of filter coefficients `coefficients` must be located in Program Memory (PM) data space so that the single cycle dual memory fetch of the processor can be used. The argument `num_coeffs` defines the number of coefficients, which must be stored in reverse order. Thus `coefficients[0]` contains the last filter coefficient, and `coefficients[num_coeffs-1]` contains the first.

The delay line has the size `num_coeffs + 1`. Before the first call, all elements must be set to zero. The first element in the delay line holds the read/write pointer being used by the function to mark the next location in the delay line to write to. The pointer should not be modified outside this function. It is needed to support the restart facility, whereby the function can be called repeatedly, carrying over previous input samples using the delay line.

The `fir_decima` function returns the address of the output array.

The library function uses the architecture's dual-data move instruction to provide simultaneous access to the filter coefficients (in PM data memory) and the delay line. When running on an ADSP-21367, ADSP-21368, ADSP-21369, ADSP-21371, or ADSP-21375 processor, the filter coefficients and the delay line must not both be allocated in external memory; otherwise, the function can generate an incorrect set of results. This occurs because in a dual-data move instruction, the hardware does not support both memory accesses allocated to external memory. Therefore, ensure that the filter coefficients or the delay line (or, optionally, both) are allocated in internal memory when running on one of the ADSP-213xx processors specified above.

Algorithm

The following equation is the basis for the algorithm:

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

where:

i = {0, 1, .., `num_output_samples`-1}
n = `num_output_samples`
k = `num_coeffs`
l = `decimation_index`

Error Conditions

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

Example

```
#include <filter.h>

#define N_DECIMATION    4
#define N_SAMPLES_OUT   128
#define N_SAMPLES_IN    (N_SAMPLES_OUT * N_DECIMATION)
#define N_COEFFS        33

float input[N_SAMPLES_IN];
float output[N_SAMPLES_OUT];
float delay[N_COEFFS + 1];
float pm coeffs[N_COEFFS];
int i;

/* Initialize the delay line */
for (i = 0; i < (N_COEFFS + 1); i++)
    delay[i] = 0.0F;

fir_decima(input, output, coeffs, delay,
           N_SAMPLES_OUT, N_COEFFS, N_DECIMATION);
```

See Also

fir, fir_interp

The `fir_decima` function makes use of circular buffers. Refer to the "Interrupts and Circular Buffering" section of Chapter 1, "Compiler", in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* for information on interrupt dispatcher considerations when circular buffers are used within an application.

fir_interp

FIR interpolation filter

Synopsis

```
#include <filter.h>

float *fir_interp (const float     input[],
                   float           output[],
                   const float pm  coefficients[],
                   float           delay[],
                   int             num_input_samples,
                   int             num_coeffs,
                   int             interp_index);
```

Description

The `fir_interp` function implements a finite impulse response (FIR) filter defined by the coefficients and the delay line supplied in the call of `fir_interp`. It generates the interpolated filtered response of the input data `input` and stores the result in the output vector `output`. To boost the signal power, the filter response is multiplied by the interpolation index `interp_index` before it is stored in the output array.

The number of input samples is specified by the argument `num_input_samples`. The size of the output vector should be `num_input_samples*interp_index`, where `interp_index` represents the interpolation index.

The array of filter coefficients `coefficients` must be located in Program Memory data space (PM) so that the single-cycle dual-memory fetch of the processor can be used. The array must contain `interp_index` sets of polyphase coefficients, where the number of polyphases in the filter is equal to the interpolation index. The number of coefficients per polyphase is specified by the argument `num_coeffs`, and therefore the total length of the array `coefficients` is of size `num_coeffs*interp_index`.

The `fir_interp` function assumes that the filter coefficients will be stored in the following order:

`coefficients[coeffs` for 1st polyphase in reverse order

`coeffs` for 2nd polyphase in reverse order

. . . .

`coeffs` for interp_index'th polyphase in reverse order]

The following example shows how the filter coefficients should be ordered for the simple case when the interpolation index is set to 1, and when the number of coefficients is 12. (Note that an interpolation index of 1 implies no interpolation, and that in this case the order of the coefficients is the same order as used by the `fir` and `fir_decima` functions).

```
c11,c10,c9,c8,c7,c6,c5,c4,c3,c2,c1,c0
```

If the interpolation index is set to 3, then the above set of coefficients should be re-ordered into three sets of polyphase coefficients in reverse order as follows

```
c9,c6,c3,c0, c10,c7,c4,c1, c11,c8,c5,c2
```

where the 1st set of polyphase coefficients `c9`, `c6`, `c3`, and `c0` are used to compute `output[k]`, the 2nd set of polyphase coefficients `c10`, `c7`, `c4`, and `c1` are used to compute `output[k+1]`, and the 3rd set of polyphase coefficients `c11`, `c8`, `c5`, and `c2` are used to compute `output[k+2]`.

In general, the re-ordering can be expressed by the following formula:

```
npoly = interp_index;
for (np = 1, i = (num_coeffs*npoly); np <= npoly; np++)
     for (nc = 1; nc <= (num_coeffs; nc++)
          coeffs[--i] = filter_coeffs[(nc * npoly) - np];
```

where `filter_coeffs[]` represents the normal order coefficients.

The delay line has the size `num_coeffs` + 1. Before the first call, all elements must be set to zero. The first element in the delay line contains the read/write pointer used by the function to mark the next location in the delay line to write to. The pointer should not be modified outside this function. It is needed to support the restart facility, whereby the function can be called repeatedly, carrying over previous input samples using the delay line.

The `fir_interp` function returns the address of the output array.

The library function uses the architecture's dual-data move instruction to provide simultaneous access to the filter coefficients (in PM data memory) and the delay line. When running on an ADSP-21367, ADSP-21368, ADSP-21369, ADSP-21371, or ADSP-21375 processor, the filter coefficients and the delay line must not both be allocated in external memory; otherwise, the function can generate an incorrect set of results. This occurs because in a dual-data move instruction, the hardware does not support both memory accesses allocated to external memory. Therefore, ensure that the filter coefficients or the delay line (or, optionally, both) are allocated in internal memory when running on one of the ADSP-213xx processors specified above.

Algorithm

The algorithm for this function is given by:

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

where:

i = {0, 1, 2, ..., `num_input_samples`-1}
m = {0, 1, 2, ..., `interp_index`-1}
n = `num_input_samples`
p = `interp_index`
k = `num_coeffs`

Error Conditions

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

Example

```
#include <filter.h>

#define N_INTERP           4
#define N_POLYPHASES       (N_INTERP)
#define N_SAMPLES_IN       128
#define N_SAMPLES_OUT      (N_SAMPLES_IN * N_INTERP)
#define N_COEFFS_PER_POLY  33
#define N_COEFFS           (N_COEFFS_PER_POLY * N_POLYPHASES)

float input[N_SAMPLES_IN];
float output[N_SAMPLES_OUT];
float delay[N_COEFFS_PER_POLY + 1];
```

```
/* Coefficients in normal order */
float filter_coeffs[N_COEFFS];

/* Coefficients in implementation order */
float pm coeffs[N_COEFFS];
int i, nc, np, scale;

/* Initialize the delay line */
for (i = 0; i < (N_COEFFS_PER_POLY + 1); i++)
    delay[i] = 0.0F;

/* Transform the normal order coefficients from a filter design
   tool into coefficients for the fir_interp function */
i = N_COEFFS;
for (np = 1, np <= N_POLYPHASES; np++)
    for (nc = 1; nc <= (N_COEFFS_PER_POLY); nc++)
        coeffs[--i] = filter_coeffs[(nc * N_POLYPHASES) - np];

fir_interp (input, output, coeffs, delay,
           N_SAMPLES_IN, N_COEFFS_PER_POLY, N_INTERP);

/* Adjust output */
scale = N_INTERP;
for (i = 0; i < N_SAMPLES_OUT; i++)
    output[i] = output[i] / scale;
```

See Also

fir, fir_decima

The `fir_interp` function makes use of circular buffers. Refer to the “Interrupts and Circular Buffering” section of Chapter 1, “Compiler”, in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* for information on interrupt dispatcher considerations when circular buffers are used within an application.

fmax

Float maximum

Synopsis

```
#include <math.h>

float fmaxf (float x, float y);
double fmax (double x, double y);
long double fmaxd (long double x, long double y);
```

Description

The `fmax` functions return the larger of their two arguments.

The `fmaxf` function is a built-in function which is implemented with an `Fn=MAX(Fx,Fy)` instruction. The `fmax` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

The `fmax` 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, lmax, lmin, max, min

fmin

Float minimum

Synopsis

```
#include <math.h>

float fminf (float x, float y);
double fmin (double x, double y);
long double fmind (long double x, long double y);
```

Description

The `fmin` 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. The `fmin` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

The `fmin` 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, lmax, lmin, max, min

gen_bartlett

Generate Bartlett window

Synopsis

```
#include <window.h>

void gen_bartlett (float dm w[],
                   int a,
                   int N);
```

Description

The `gen_bartlett` function generates a vector containing the Bartlett window. The length is specified by parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

The Bartlett window is similar to the triangle window (see "gen_triangle" on page 2-165) but has the following different properties:

- The Bartlett window 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 an `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.

Algorithm

The algorithm for this function is given by:

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

where

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

Domain

`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_vonhann

gen_blackman

Generate Blackman window

Synopsis

```
#include <window.h>

void gen_blackman (float dm w[],
                   int a,
                   int N);
```

Description

The `gen_blackman` function generates a vector containing the Blackman window. The length of the required window is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

Algorithm

The algorithm for this function is given by:

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

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

Domain

`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_vonhann

gen_gaussian

Generate Gaussian window

Synopsis

```
#include <window.h>

void gen_gaussian (float dm w[],
                   float    alpha,
                  int a,
                  int N);
```

Description

The `gen_gaussian` function generates a vector containing the Gaussian window. The length of the required window is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

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

Algorithm

The algorithm for this function is given by:

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

where

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

Domain

`a > 0; N > 0; a > 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_vonhann

gen_hamming

Generate Hamming window

Synopsis

```
#include <window.h>

void gen_hamming (float dm w[],
                  int a,
                  int N);
```

Description

The `gen_hamming` function generates a vector containing the Hamming window. The length of the required window is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

Algorithm

The algorithm for this function is given by:

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

where

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

Domain

```
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_vonhann

gen_hanning

Generate Hanning window

Synopsis

```
#include <window.h>

void gen_hanning (float dm w[],
                  int a,
                  int N);
```

Description

The `gen_hanning` function generates a vector containing the Hanning window. The length of the required window is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`. This window is also known as the Cosine window.

Algorithm

The following equation is the basis of the algorithm.

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

where:

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

Domain

`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_vonhann

gen_harris

Generate Harris window

Synopsis

```
#include <window.h>

void gen_harris (float dm w[],
                 int a,
                 int N);
```

Description

The gen_harris function generates a vector containing the Harris window. The length of the required window is specified by the parameter N, and the stride parameter a is used to space the window values within the output vector w. The length of the output vector should therefore be N*a. This window is also known as the Blackman-Harris window.

Algorithm

The following equation is the basis of the algorithm.

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

where:

N = window_size
w = harris_window
n = {0, 1, 2, ..., N-1}

Domain

```
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_vonhann

gen_kaiser

Generate Kaiser window

Synopsis

```
#include <window.h>

void gen_kaiser (float dm w[],
                 int a,
                 int N);
```

Description

The `gen_kaiser` function generates a vector containing the Kaiser window. The length of the required window is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`. The b value is specified by parameter `beta`.

Algorithm

The following equation is the basis of the algorithm.

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

where:

N = `window_size`
w = `kaiser_window`
n = {0, 1, 2, ..., `N-1`}
α = `(N - 1) / 2`
$I_0(\beta)$ represents the zeroth-order modified Bessel function of the first kind

Domain

`a > 0; N > 0; b > 0.0`

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_vonhann

gen_rectangular

Generate rectangular window

Synopsis

```
#include <window.h>

void gen_rectangular (float dm w[],
                      int a,
                      int N);
```

Description

The `gen_rectangular` function generates a vector containing the rectangular window. The length of the required window is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

Algorithm

$w[n] = 1$

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

Domain

`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_vonhann

gen_triangle

Generate triangle window

Synopsis

```
#include <window.h>

void gen_triangle (float dm w[],
                   int a,
                   int N);
```

Description

The `gen_triangle` function generates a vector containing the triangle window. The length of the required window is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

Refer to the Bartlett window (described on page 2-149) regarding the relationship between it and the triangle window.

Algorithm

For even `n`, the following equation applies.

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

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

For odd n, the following equation applies.

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

where

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

Domain

`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, gen_vonhann

gen_vonhann

Generate von Hann window

Synopsis

```
#include <window.h>

void gen_vonhann (float dm w[],
                  int a,
                  int N);
```

Description

The `gen_vonhann` function is identical to `gen_hanning` window (described on page 2-157).

Error Conditions

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

See Also

gen_hanning

histogram

Histogram

Synopsis

```
#include <stats.h>

int *histogram (int out[],
                const int 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, 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.

For maximal performance, this function does not perform out-of-bounds checking. Therefore, all values within the input array must be within range (that is, between 0 and `bin_size * out_len`).

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 processor idle instruction

Synopsis

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

Description

The `idle` function invokes the processor's `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, refer to the appropriate SHARC processor hardware reference manual.

Error Conditions

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

Example

```
#include <processor_include.h>

idle ();
```

See Also

interrupt, signal

ifft

Inverse complex radix-2 Fast Fourier Transform

Synopsis

```
#include <filter.h>

complex_float  *ifft (complex_float        dm input[],
                      complex_float        dm temp[],
                      complex_float        dm output[],
                      const complex_float  pm twiddle[],
                      int                  twiddle_stride,
                      int                  n );
```

Description

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

The size of the input array `input`, the output array `output`, and the temporary working buffer `temp` must be at least `n`, where `n` represents the number of points in the FFT; `n` must be a power of 2 and no smaller than 8. If the input data can be overwritten, memory can be saved by setting the pointer of the temporary array explicitly to the input array, or to `NULL`. (In either case the input array will also be used as the temporary working array.)

The minimal size of the twiddle table must be `n`/2. A larger twiddle table may be used provided that the value of the twiddle table stride argument `twiddle_stride` is set appropriately. If the size of the twiddle table is `x`, then `twiddle_stride` must be set to (2*`x`)/`n`.

The library function `twidfft` (on page 2-258) can be used to compute the required twiddle table. The coefficients generated are positive cosine coefficients for the real part and negative sine for the imaginary part.

> ℹ For the SHARC 21xxx SIMD processors, the library also contains the `ifftf` function (see "ifftf (SHARC SIMD Processors)" on page 2-174), which is an optimized implementation of an inverse complex FFT using a fast radix-2 algorithm. The `ifftf` function, however, imposes certain memory alignment requirements that may not be appropriate for some applications.

The function returns the address of the output array.

Algorithm

The following equation is the basis of the algorithm.

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

Error Conditions

The `ifft` function does not return any error condition.

Example

```
#include <filter.h>

#define N_FFT   64

complex_float  input[N_FFT];
complex_float  output[N_FFT];
complex_float  temp[N_FFT];

int            twiddle_stride = 1;
complex_float  pm twiddle[N_FFT/2];
```

```
   /* Populate twiddle table */
twidfft(twiddle, N_FFT);

   /* Compute Fast Fourier Transform */
ifft(input, temp, output, twiddle, twiddle_stride, N_FFT);
```

See Also

cfft, ifftf (SHARC SIMD Processors), ifftN (SHARC SIMD Processors), rfft, twidfft

ⓘ For the ADSP-21xxx SIMD processors the `ifft` function uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

ⓘ The `ifft` function makes use of circular buffers. Refer to the "Interrupts and Circular Buffering" section of Chapter 1, "Compiler", in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* for information on interrupt dispatcher considerations when circular buffers are used within an application.

ifftf (SHARC SIMD Processors)

Fast inverse complex radix-2 Fast Fourier Transform

Synopsis

```
#include <filter.h>

void ifftf (float data_real[], float data_imag[],
            float temp_real[], float temp_imag[],
            const float twid_real[],
            const float twid_imag[],
            int n);
```

Description

The ifftf function transforms the frequency domain complex input signal sequence to the time domain by using the accelerated version of the Discrete Fourier Transform known as a Fast Fourier Transform or FFT. It decimates in frequency, using an optimized radix-2 algorithm.

The array data_real contains the real part of a complex input signal, and the array data_imag contains the imaginary part of the signal. On output, the function overwrites the data in these arrays and stores the real part of the inverse FFT in data_real, and the imaginary part of the inverse FFT in data_imag. If the input data is to be preserved, it must first be copied to a safe location before calling this function. The argument n represents the number of points in the inverse FFT. It must be a power of 2 and must be at least 64.

The ifftf function has been designed for optimal performance and requires that the arrays data_real and data_imag are aligned on an address boundary that is a multiple of the FFT size. For certain applications, this alignment constraint may not be appropriate; in such cases, the application should call the ifft function instead with no loss of facility (apart from performance).

The arrays temp_real and temp_imag are used as intermediate temporary buffers and should each be of size n.

The twiddle table is passed in using the arrays twid_real and twid_imag. The array twid_real contains the positive cosine factors, and the array twid_imag contains the negative sine factors. Each array should be of size n/2. The twidfftf function (on page 2-261) may be used to initialize the twiddle table arrays.

It is recommended that the arrays containing real parts (data_real, temp_real, and twid_real) are allocated in separate memory blocks from the arrays containing imaginary parts (data_imag, temp_imag, and twid_imag). Otherwise, the performance of the function degrades.

The ifftf function has been implemented to make highly efficient use of the processor's SIMD capabilities and long word addressing mode. The function therefore imposes the following restrictions:

- All the arrays that are passed to the function must be allocated in internal memory. The DSP run-time library does not contain a version of the function that can be used with data in external memory.
- The function should not be used with any application that relies on the -reserve register[, register...] switch

For more information, refer to refer to "Implications of Using SIMD Mode" and "Using Data in External Memory".

Error Conditions

The ifftf function does not return an error condition.

Example

```
#include <filter.h>

#define FFT_SIZE 1024

#pragma align 1024
float dm input_r[FFT_SIZE];
#pragma align 1024
float pm input_i[FFT_SIZE];

float dm temp_r[FFT_SIZE];
float pm temp_i[FFT_SIZE];
float dm twid_r[FFT_SIZE/2];
float pm twid_i[FFT_SIZE/2];

twidfftf(twid_r,twid_i,FFT_SIZE);
ifftf(input_r,input_i,
      temp_r,temp_i,
      twid_r,twid_i,FFT_SIZE);
```

See Also

cfftf (SHARC SIMD Processors), ifft, ifftN (SHARC SIMD Processors), rfftf_2 (SHARC SIMD Processors), twidfftf (SHARC SIMD Processors)

ifftN

N-point inverse complex radix-2 Fast Fourier Transform

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 `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 fourteen distinct functions in this set. 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` are two floating-point arrays of N points. The array `real_input` contains the real components of the inverse FFT input and the array `imag_input` contains the imaginary components.

If there are fewer than N actual data points, you must pad the arrays 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 time-domain signal generated by the `ifftN` functions is stored in the arrays `real_output` and `imag_output`. The array `real_output` contains the real component of the complex output signal, and the array `imag_output` contains the imaginary component. The output is scaled by `N`, the number of points in the inverse FFT. The functions return a pointer to the `real_output` array.

If the input data can be overwritten, then the `ifftN` functions allow the array `real_input` to share the same memory as the array `real_output`, and `imag_input` to share the same memory as `imag_output`. This improves memory usage, but at the cost of run-time performance.

ⓘ These library functions have not been optimized for SHARC SIMD processors. Applications that run on SHARC SIMD processors should use the FFT functions that are defined in the header file `filter.h`, and described under cfftN (SHARC SIMD Processors) instead.

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];

ifft2048 (real_input, imag_input, real_output, imag_output);
```

See Also

cfftN, ifft, ifftN (SHARC SIMD Processors), rfftN

The `ifftN` functions make use of circular buffers. Refer to the "Interrupts and Circular Buffering" section of Chapter 1, "Compiler", in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* for information on interrupt dispatcher considerations when circular buffers are used within an application.

ifftN (SHARC SIMD Processors)

N-point inverse complex radix-2 Fast Fourier Transform

Synopsis

```
#include <filter.h>

complex_float *ifft65536 (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft32768 (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft16384 (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft8192  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft4096  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft2048  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft1024  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft512   (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft256   (complex_float dm input[],
                          complex_float dm output[]);
```

```
complex_float *ifft128   (complex_float input[],
                          complex_float dm output[]);

complex_float *ifft64    (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft32    (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft16    (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft8     (complex_float dm input[],
                          complex_float dm output[]);
```

Description

These `ifftN` functions are defined in the header file filter.h; they have been optimized to take advantage of the SIMD capabilities of the ADSP-211xx, ADSP-212xx, ADSP-213xx, and ADSP-214xx processors. Therefore, they are not supported by the ADSP-210xx processor family. These FFT functions require complex arguments to ensure that the real and imaginary parts are interleaved in memory and are thus accessible in a single cycle, using the wider data bus of the processor.

Each of these `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 fourteen distinct functions in this set. 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 (input, output);`

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. 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. Optimal memory usage can be achieved by specifying the input array as the output array, but at the cost of run-time performance.

The `ifftN` functions return a pointer to the `output` array.

The `ifftN` functions use the input array as an intermediate workspace. If the input data is to be preserved it must first be copied to a safe location before calling these functions.

Error Conditions

The `ifftN` functions do not return error conditions.

Example

```
#include <filter.h>
#define N 2048
complex_float input[N], output[N];

ifft2048 (input, output);
```

See Also

cfftN (SHARC SIMD Processors), ifft, ifftf (SHARC SIMD Processors), rfftN (SHARC SIMD Processors)

By default, these functions use SIMD. Refer to “Implications of Using SIMD Mode” on page 2-23 for more information.

iir

Infinite impulse response (IIR) filter

Synopsis (Scalar-Valued Version)

```
#include <filters.h>

float iir (float sample,
           const float  pm a_coeffs[],
           const float  pm b_coeffs[],
           float        dm state[],
           int             taps);
```

Synopsis (Vector-Valued Version)

ADSP-210xx Processors

```
#include <filter.h>

float *iir_vec (const float  dm input[],
                float        dm output[],
                const float  pm coeffs[],
                float        dm state[],
                int             samples,
                int             sections);
```

ADSP-21xxx SIMD Processors

```
#include <filter.h>

float *iir (const float  dm input[],
                float        dm output[],
                const float  pm coeffs[],
                float        dm state[],
                int             samples,
                int             sections);
```

Description (Scalar-Valued Version)

The scalar-valued version of the iir function implements a parallel second-order direct form II infinite impulse response (IIR) filter. The function returns the filtered response of the input data sample. The characteristics of the filter are dependent upon a set of coefficients, a delay line, and the length of the filter. The length of filter is specified by the argument taps.

The set of IIR filter coefficients is composed of a-coefficients and b-coefficients. The a0 coefficient is assumed to be 1.0, and the remaining a-coefficients should be scaled accordingly and stored in the array a_coeffs in reverse order. The length of the a_coeffs array is taps and therefore a_coeffs[0] should contain ataps, and a_coeffs[taps-1] should contain a1.

The b-coefficients are stored in the array b_coeffs, also in reverse order. The length of the b_coeffs is taps+1, and so b_coeffs[0] contains btaps and b_coeffs[taps] contains b0.

Both the a_coeffs and b_coeffs arrays must be located in Program Memory (PM) so that the single-cycle dual-memory fetch of the processor can be used.

Each filter should have its own delay line which the function maintains in the array state. The array should be initialized to zero before calling the function for the first time and should not be modified by the calling program. The length of the state array should be taps+1 as the function uses the array to store a pointer to the current delay line.

The library function uses the architecture's dual-data move instruction to provide simultaneous access to the filter coefficients (in PM data memory) and the delay line. When running on an ADSP-21367, ADSP-21368, ADSP-21369, ADSP-21371, or ADSP-21375 processor, the filter coefficients and the delay line must not both be allocated in external memory; otherwise, the function can generate an incorrect set of results. This occurs because in a dual-data move instruction, the hardware does not support both memory accesses allocated to external memory. Therefore, ensure that the filter coefficients or the delay line (or, optionally, both) are allocated in internal memory when running on one of the ADSP-213xx processors specified above.

The flow graph (Figure 2-2) corresponds to the `iir()` routine as part of the DSP run-time library.

The `biquad` function should be used instead of the `iir` function if a multi-stage filter is required.

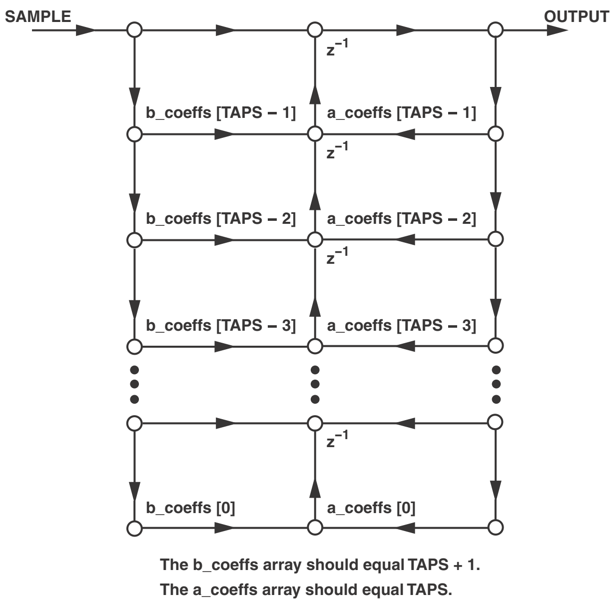

Figure 2-2. Flow Graph

Description (Vector-Valued Version)

The vector-valued versions of the `iir` function implement an infinite impulse response (IIR) filter defined by the coefficients and delay line that are supplied in the call to the function. The filter is implemented as a cascaded biquad, and generate the filtered response of the input data `input` and store the result in the output vector `output`. The number of input samples and the length of the output vector is specified by the argument `samples`.

The characteristics of the filter are dependent upon the filter coefficients and the number of biquad sections. The number of sections is specified by the argument `sections`, and the filter coefficients are supplied to the function using the argument `coeffs`. Each stage has four coefficients which must be ordered in the following form:

```
[a2 stage 1, a1 stage 1, b2 stage 1, b1 stage 1, a2 stage 2, ...]
```

The function assumes that the value of `B0` is `1.0`, and so the `B1` and `B2` coefficients should be scaled accordingly. As a consequence of this, all the output generated by the `iir` function must be scaled by the product of all the `B0` coefficients to obtain the correct signal amplitude. The function also assumes that the value of the `A0` coefficient is `1.0`, and the `A1` and `A2` coefficients should be normalized. The `A1` and `A2` coefficients should be negated if they have been imported from most filter design tools. These requirements are demonstrated in the example below.

The `coeffs` array must be allocated in Program Memory (PM) as the function uses the single-cycle dual-memory fetch of the processor. The definition of the `coeffs` array is therefore:

```
float pm coeffs[4*sections];
```

Each filter should have its own delay line which is represented by the array `state`. The `state` array should be large enough for two delay elements per biquad section and hold an internal pointer that allows the filter to be restarted. The definition of the state is:

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

The `state` array should be initially cleared to zero before calling the function for the first time and should not be modified by the user program.

The function returns a pointer to the output vector.

The vector-valued versions of the `iir` functions are based on the following algorithm:

$$H(z) = \prod_{n=0}^{sections-1} \frac{1 + \left(\frac{b_n 1}{b_n 0}\right) z^{-1} + \left(\frac{b_n 2}{b_n 0}\right) z^{-2}}{1 - \left(\frac{a_n 1}{a_n 0}\right) z^{-1} - \left(\frac{a_n 2}{a_n 0}\right) z^{-2}}$$

To get the correct amplitude of the signal, H(z) should be adjusted by this formula:

$$H(z) = H(z) \bullet \left(\prod_{n=0}^{sections-1} \frac{b_n 0}{a_n 0} \right)$$

Error Conditions

The `iir` functions do not return an error condition.

Example

Scalar-Valued

```
#include <filters.h>

#define NSAMPLES 256
#define TAPS 10

float input[NSAMPLES];
float output[NSAMPLES];
float pm a_coeffs[TAPS];
float pm b_coeffs[TAPS+1];

float state[TAPS + 1];
int i;

for (i = 0; i < TAPS+1; i++)
    state[i] = 0;

for (i = 0; i < NSAMPLES; i++)
   output[i] = iir (input[i], a_coeffs, b_coeffs, state, TAPS);
```

Vector-Valued

```
#include <filter.h>

#define SAMPLES 100
#define SECTIONS 4
```

```
/* Coefficients generated by a filter design tool that uses
   a direct form II */

const struct {
   float a0;
   float a1;
   float a2;
} A_coeffs[SECTIONS];

const struct {
   float b0;
   float b1;
   float b2;
} B_coeffs[SECTIONS];

/* Coefficients for the iir function */

float pm coeffs[4 * SECTIONS];

/* Input, Output, and State Arrays */

float input[SAMPLES], output[SAMPLES];
float state[2*SECTIONS + 1];

float scale;      /* used to scale the output from iir */

/* Utility Variables */
float a0,a1,a2;
float b0,b1,b2;
int i;
```

```
/* Transform the A-coefficients and B-coefficients from a filter
   design tool into coefficients for the iir function */

scale = 1.0;

for (i = 0; i < SECTIONS; i++) {

   a0 = A_coeffs[i].a0;
   a1 = A_coeffs[i].a1;
   a2 = A_coeffs[i].a2;

   /* Negate A1 and A2 (not reqd for all filter design tools) */

   a1 = -a1;
   a2 = -a2;

   coeffs[(i*4) + 0] = (a2/a0);
   coeffs[(i*4) + 1] = (a1/a0);

   b0 = B_coeffs[i].b0;
   b1 = B_coeffs[i].b1;
   b2 = B_coeffs[i].b2;

   coeffs[(i*4) + 2] = (b2/b0);
   coeffs[(i*4) + 3] = (b1/b0);

   scale = scale * (b0/a0);
}
```

```
/* Call the iir function */

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

#if defined(__SIMDSHARC__)
  iir (input, output, coeffs, state, SAMPLES, SECTIONS);
#else
  iir_vec (input, output, coeffs, state, SAMPLES, SECTIONS);
#endif

/* Adjust output by all (b0/a0) terms */

for (i = 0; i < SAMPLES; i++)
   output[i] = output[i] * scale;
```

See Also

biquad, fir

The `iir_vec` function makes use of circular buffers. Refer to the "Interrupts and Circular Buffering" section of Chapter 1, "Compiler" in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* for information on interrupt dispatcher considerations when circular buffers are used within an application.

matinv

Real matrix inversion

Synopsis

```
#include <matrix.h>

float *matinvf (float dm *output,
                const float dm *input, int samples);

double *matinv (double dm *output,
                const double dm *input, int samples);

long double *matinvd (long double dm *output,
                      const long double dm *input, int samples);
```

Description

The `matinv` functions employ Gauss-Jordan elimination with full pivoting to compute the inverse of the input matrix `input` and store the result in the matrix `output`. The dimensions of the matrices `input` and `output` are `[samples][samples]`. The functions return a pointer to the output matrix.

Error Conditions

If no inverse exists for the input matrix, the functions return a null pointer.

Example

```
#include <matrix.h>
#define N 8

double a[N][N];
double a_inv[N][N];

matinv ((double *)(a_inv),(double *)(a),N);
```

See Also

No related function.

matmadd

Real matrix + matrix addition

Synopsis

```
#include <matrix.h>

float *matmaddf (float dm *output,
                 const float dm *a,
                 const float dm *b, int rows, int cols);

double *matmadd (double dm *output,
                 const double dm *a,
                 const double dm *b, int rows, int cols);

long double *matmaddd (long double dm *output,
               const long double dm *a,
               const long double dm *b, int rows, int cols);

float *matadd  (float dm *output,
               const float dm *a,
               const float dm *b, int rows, int cols);
```

Description

The `matmadd` functions perform a matrix addition of the input matrices `a[][]` and `b[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]`, `b[rows][cols]`, and `output[rows][cols]`.

The functions return a pointer to the output matrix.

The `matadd` function is equivalent to `matmaddf` and is provided for compatibility with previous versions of VisualDSP++.

Error Conditions

The `matmadd` functions do not return an error condition.

Example

```
#include <matrix.h>

#define ROWS 4
#define COLS 8

double input_1[ROWS][COLS], *a_p = (double *) (&input_1);
double input_2[ROWS][COLS], *b_p = (double *) (&input_2);
double result[ROWS][COLS], *res_p = (double *) (&result);

matmadd (res_p, a_p, b_p, ROWS, COLS);
```

See Also

cmatmadd, matmmlt, matmsub, matsadd

For the ADSP-21xxx SIMD processors the `matmaddf` function (and `matmadd`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

matmmlt

Real matrix * matrix multiplication

Synopsis

```
#include <matrix.h>

float *matmmltf (float dm *output,
                 const float dm *a,
                 const float dm *b,
                 int a_rows, int a_cols, b_cols);

double *matmmlt (double dm *output,
                 const double dm *a,
                 const double dm *b,
                 int a_rows, int a_cols, b_cols);

long double *matmmltd (long double dm *output,
                 const long double dm *a,
                 const long double dm *b,
                 int a_rows, int a_cols, b_cols);

float *matmul (float dm *output,
                 const float dm *a,
                 const float dm *b,
                 int a_rows, int a_cols, b_cols);
```

Description

The `matmmlt` functions perform a matrix multiplication of the input matrices `a[][]` and `b[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[a_rows][a_cols]`, `b[a_cols][b_cols]`, and `output[a_rows][b_cols]`.

The functions return a pointer to the output matrix.

The `matmult` function is equivalent to `matmul` and is provided for compatibility with previous versions of VisualDSP++.

Algorithm

The following equation is the basis of the algorithm.

$$c_{i,j} = \sum_{l=0}^{a\_cols-1} a_{i,l} \bullet b_{l,j}$$

where

i = {0, 1, 2, ..., `a_rows`-1}, j = {0, 1, 2, ..., `b_cols`-1}

Error Conditions

The `matmmlt` functions do not return an error condition.

Example

```
#include <matrix.h>

#define ROWS_1 4
#define COLS_1 8
#define COLS_2 2

double input_1[ROWS_1][COLS_1], *a_p = (double *) (&input_1);
double input_2[COLS_1][COLS_2], *b_p = (double *) (&input_2);
double result[ROWS_1][COLS_2], *res_p = (double *) (&result);

matmmlt (res_p, a_p, b_p, ROWS_1, COLS_1, COLS_2);
```

See Also

cmatmmlt, matmadd, matmsub, matsmlt

The `matmmlt` functions make use of circular buffers. Refer to the "Interrupts and Circular Buffering" section of Chapter 1, "Compiler" in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* for information on interrupt dispatcher considerations when circular buffers are used within an application.

matmsub

Real matrix – matrix subtraction

Synopsis

```
#include <matrix.h>

float *matmsubf (float dm *output,
                const float dm *a,
                const float dm *b, int rows, int cols);

double *matmsub (double dm *output,
                const double dm *a,
                const double dm *b, int rows, int cols);

long double *matmsubd (long double dm *output,
                const long double dm *a,
                const long double dm *b, int rows, int cols);

float *matsub  (float dm *output,
                const float dm *a,
                const float dm *b, int rows, int cols);
```

Description

The `matmsub` functions perform a matrix subtraction of the input matrices `a[][]` and `b[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]`, `b[rows][cols]`, and `output[rows][cols]`.

The functions return a pointer to the output matrix.

The `matsub` function is equivalent to `matmsubf` and is provided for compatibility with previous versions of VisualDSP++.

Error Conditions

The `matmsub` functions do not return an error condition.

Example

```
#include <matrix.h>

#define ROWS 4
#define COLS 8

double input_1[ROWS][COLS], *a_p = (double *) (&input_1);
double input_2[ROWS][COLS], *b_p = (double *) (&input_2);
double result[ROWS][COLS],  *res_p = (double *) (&result);

matmsub (res_p, a_p, b_p, ROWS, COLS);
```

See Also

cmatmsub, matmadd, matmmlt, matssub

For the ADSP-21xxx SIMD processors the `matmsubf` function (and `matmsub`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to “Implications of Using SIMD Mode” on page 2-23 for more information.

matsadd

Real matrix + scalar addition

Synopsis

```
#include <matrix.h>

float *matsaddf (float dm *output, const float dm *a,
                 float scalar, int rows, int cols);

double *matsadd (double dm *output, const double dm *a
                 double scalar, int rows, int cols);

long double *matsaddd (long double dm *output,
                       const long double dm *a,
                       long double scalar, int rows, int cols);
```

Description

The `matsadd` functions add a scalar to each element of the input matrix `a[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The `matsadd` functions do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS 4
#define COLS 8

double input[ROWS][COLS], *a_p = (double *) (&input);
double result[ROWS][COLS], *res_p = (double *) (&result);
double x;

matsadd (res_p, a_p, x, ROWS, COLS);
```

See Also

cmatsadd, matmadd, matsmlt, matssub

For the ADSP-21xxx SIMD processors the `matsaddf` function (and `matsadd`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

matsmlt

Real matrix * scalar multiplication

Synopsis

```
#include <matrix.h>

float *matsmltf (float dm *output, const float dm *a,
                 float scalar, int rows, int cols);

double *matsmlt (double dm *output, const double dm *a
                 double scalar, int rows, int cols);

long double *matsmltd (long double dm *output,
                       const long double dm *a,
                       long double scalar, int rows, int cols);

float *matscalmult (float dm *output, const float dm *a,
                    float scalar, int rows, int cols);
```

Description

The `matsmlt` functions multiply a scalar with each element of the input matrix `a[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`.

The functions return a pointer to the output matrix.

The `matscalmult` function is equivalent to `matsmltf` and is provided for compatibility with previous versions of VisualDSP++.

Error Conditions

The `matsmlt` functions do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS 4
#define COLS 8

double input[ROWS][COLS], *a_p = (double *) (&input);
double result[ROWS][COLS], *res_p = (double *) (&result);
double x;

matsmlt (res_p, a_p, x, ROWS, COLS);
```

See Also

cmatsmlt, matmmlt, matsadd, matssub

For the ADSP-21xxx SIMD processors the `matsmltf` function (and `matsmlt`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

matssub

Real matrix – scalar subtraction

Synopsis

```
#include <matrix.h>

float *matssubf (float dm *output, const float dm *a,
                 float scalar, int rows, int cols);

double *matssub (double dm *output, const double dm *a
                 double scalar, int rows, int cols);

long double *matssubd (long double dm *output,
                       const long double dm *a,
                       long double scalar, int rows, int cols);
```

Description

The `matssub` functions subtract a scalar from each element of the input matrix `a[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The `matssub` functions do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS 4
#define COLS 8

double input[ROWS][COLS], *a_p = (double *) (&input);
double result[ROWS][COLS], *res_p = (double *) (&result);
double x;
matssub (res_p, a_p, x, ROWS, COLS);
```

See Also

cmatssub, matmsub, matsadd, matsmlt

For the ADSP-21xxx SIMD processors the `matssubf` function (and `matssub`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

mean

Mean

Synopsis

```
#include <stats.h>

float meanf (const float in[], int length);
double mean (const double in[], int length);
long double meand (const long double in[], int length);
```

Description

The `mean` functions return the mean of the input array `in[]`. The length of the input array is `length`.

Error Conditions

The `mean` functions do not return an error condition.

Example

```
#include <stats.h>

#define SIZE 256
double data[SIZE];
double result;
result = mean (data, SIZE);
```

See Also

var

For the ADSP-21xxx SIMD processors the `meanf` function (and `mean`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

mu_compress

µ-law compression

Synopsis (Scalar-Valued)

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

Synopsis (Vector-Valued)

ADSP-210xx Processors

```
#include <filter.h>
int *mu_compress_vec (const int  dm input[],
                      int        dm output[],
                      int        length);
```

ADSP-21xxx SIMD Processors

```
#include <filter.h>
int *mu_compress(const int    dm input[],
                       int    dm output[],
                       int    length);
```

Description

The `mu_compress` functions take linear 14-bit speech samples and compress them according to ITU recommendation G.711 (µ-law definition).

The scalar version of `mu_compress` inputs a single data sample and returns an 8-bit compressed output sample.

The vector versions of `mu_compress` take the array `input`, and return the compressed 8-bit samples in the vector `output`. The parameter `length` defines the size of both the input and output vectors. The functions return a pointer to the output array.

The vector versions of `mu_compress` uses serial port 0 to perform the companding on an ADSP-21160 processor; therefore, serial port 0 must not be in use when this routine is called. The serial port is not used by this function on any other ADSP-21xxx SIMD architectures.

Error Conditions

The `mu_compress` functions do not return an error condition.

Example

Scalar-Valued

```
#include <comm.h>

int sample, compress;
compress = mu_compress (sample);
```

Vector-Valued

```
#include <filter.h>

#define NSAMPLES 50
int data [NSAMPLES], compressed[NSAMPLES];

#if defined(__SIMDSHARC__)
   mu_compress (data, compressed, NSAMPLES);
#else
   mu_compress_vec (data, compressed, NSAMPLES);
#endif
```

See Also

a_compress, mu_expand

mu_expand

µ-law expansion

Synopsis (Scalar-Valued)

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

Synopsis (Vector-Valued)

ADSP-210xx Processors

```
#include <filter.h>

int *mu_expand_vec (const int  dm input[],
                    int        dm output[],
                    int        length);
```

ADSP-21xxx SIMD Processors

```
#include <filter.h>

int *mu_expand(const int  dm input[],
               int        dm output[],
               int        length);
```

Description

The `mu_expand` functions take 8-bit compressed speech samples and expand them according to ITU recommendation G.711 (µ-law definition).

The scalar version of `mu_expand` inputs a single data sample and returns a linear 14-bit signed sample.

The vector version of `mu_expand` takes an array of 8-bit compressed speech samples and expands it according to ITU recommendation G.711 (µ-law definition). The array returned contains linear 14-bit signed samples. These functions returns a pointer to the output data array.

The vector versions of `mu_expand` uses serial port 0 to perform the companding on an ADSP-21160 processor. Therefore, serial port 0 must not be in use when this routine is called. The serial port is not used by this function on any other ADSP-21xxx SIMD architectures.

Error Conditions

The `mu_expand` functions do not return an error condition.

Example

Scalar-Valued

```
#include <comm.h>

int compressed_sample, expanded;
expanded = mu_expand (compressed_sample);
```

Vector-Valued

```
#include <filter.h>

#define NSAMPLES 50
int data [NSAMPLES];
int expanded_data[NSAMPLES];

#if defined(__SIMDSHARC__)
   mu_expand (data, expanded_data, NSAMPLES);
#else
   mu_expand_vec (data, expanded_data, NSAMPLES);
#endif
```

See Also

a_expand, mu_compress

norm

Normalization

Synopsis

```
#include <complex.h>

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

Description

The normalization functions normalize the complex input `a` and return the result. Normalization of a complex number is defined as:

Algorithm

The following equations are the basis of the algorithm.

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

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

Error Conditions

The normalization functions return zero if `cabs(a)` is equal to zero.

Example

```
#include <complex.h>

complex_double x = {2.0,-5.0};
complex_double z;
z = norm(x);     /* z = (0.4472,-0.8944) */
```

See Also

No related function.

polar

Construct from polar coordinates

Synopsis

```
#include <complex.h>

complex_float polarf (float mag, float phase);
complex_double polar (double mag, double phase);
complex_long_double polard (long double mag,
                            long double phase);
```

Description

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

Algorithm

The algorithm for transforming a polar coordinate into a Cartesian coordinate is:

$\mathrm{Re}(c) = mag * \cos(phase)$

$\mathrm{Im}(c) = mag * \sin(phase)$

Error Conditions

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

Example

```
#include <complex.h>

#define PI 3.14159265

float magnitude = 2.0;
float phase = PI;
complex_float z;

z = polarf (magnitude,phase);     /* z.re = -2.0, z.im = 0.0 */
```

See Also

arg, cartesian

poll_flag_in

Test input flag

Synopsis

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

Description

The `poll_flag_in` function tests the specified flag (0, 1, 2, or 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 2-24. 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 <processor_include.h>

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

See Also

interrupt, set_flag

rfft

Real radix-2 Fast Fourier Transform

Synopsis

```
#include <filter.h>

complex_float  *rfft  (float                dm input[],
                       float                dm temp[],
                       complex_float        dm output[],
                       const complex_float  pm twiddle[],
                       int                  twiddle_stride,
                       int                  n);
```

Description

The `rfft` function transforms the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT).

The size of the input array `input` and the temporary working buffer `temp` must be at least `n`, where `n` represents the number of points in the FFT; `n` must be a power of 2 and no smaller than 16. If the input data can be overwritten, memory can be saved by setting the pointer of the temporary array explicitly to the input array or to `NULL`. (In either case the input array will also be used as a temporary working array.)

As the complex spectrum of a real FFT is symmetrical about the midpoint, the `rfft` function will only generate the first `(n/2)+1` points of the FFT, and so the size of the output array output must be at least of length `(n/2) + 1`.

After returning from the `rfft` function, the output array contains the following values:

- DC component of the signal in `output[0].re (output[0].im = 0)`
- First half of the complex spectrum in `output[1] ... output[(n/2)-1]`
- Nyquist frequency in `output[n/2].re (output[n/2].im = 0)`

Refer to the Example section below to see how an application would construct the full complex spectrum, using the symmetry of a real FFT.

The minimal size of the twiddle table must be `n/2`. A larger twiddle table may be used, providing that the value of the twiddle table stride argument `twiddle_stride` is set appropriately. If the size of the twiddle table is `x`, then `twiddle_stride` must be set to `(2*x)/n`.

The library function `twidfft` (on page 2-258) can be used to compute the required twiddle table. The coefficients generated are positive cosine coefficients for the real part and negative sine coefficients for the imaginary part.

> For the ADSP-21xxx SIMD processors the library also contains the `rfftf_2` function. (For more information, see “rfftf_2 (SHARC SIMD Processors)” on page 2-227.) This function is an optimized implementation of a real FFT using a fast radix-2 algorithm, capable of computing two real FFTs in parallel. The `rfftf_2` function, however, imposes certain memory alignment requirements that may not be appropriate for some applications.

The function returns the address of the output array.

Algorithm

The following equation is the basis of the algorithm.

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

Error Conditions

The `rfft` function does not return any error condition.

Example

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

#define FFTSIZE 32

float sigdata[FFTSIZE];          /* input signal */
complex_float r_output[FFTSIZE]; /* FFT of input signal */
complex_float i_output[FFTSIZE]; /* inverse of r_output */

complex_float i_temp[FFTSIZE];   /* temporary array */
complex_float c_temp[FFTSIZE];   /* temporary array */
float *r_temp = (float *) c_temp;

complex_float pm twiddle_table[FFTSIZE/2];

int i;

/* Initialize the twiddle table */

twidfft (twiddle_table,FFTSIZE);
```

```
/* Calculate the FFT of a real signal */

rfft (sigdata,r_temp,r_output,twiddle_table,1,FFTSIZE);

    /* (rfft sets r_output[FFTSIZE/2] to the Nyquist) */

/* Add the 2nd half of the spectrum */

for (i = 1; i < (FFTSIZE/2); i++) {
    r_output[FFTSIZE - i] = conjf (r_output[i]);
}

/* Calculate the inverse of the FFT */

ifft (r_output,i_temp,i_output,twiddle_table,1,FFTSIZE);
```

See Also

cfft, fft_magnitude, ifft, rfftf_2 (SHARC SIMD Processors), rfftN (SHARC SIMD Processors), twidfft

The `rfft` function makes use of circular buffers. Refer to the "Interrupts and Circular Buffering" section of Chapter 1, "Compiler", in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* for information on interrupt dispatcher considerations when circular buffers are used within an application.

rfft_mag (SHARC SIMD Processors)

rfft magnitude

Synopsis

```
#include <filter.h>

float *rfft_mag (complex_float dm input[],
                 float dm output[],
                 int fftsize);

float *fft_mag (complex_float dm input[],
                float dm output[],
                int fftsize);
```

Description

The `rfft_mag` function computes a normalized power spectrum from the output signal generated by a `rfftN` function. The size of the signal and the size of the power spectrum is `fftsize/2`.

The function returns a pointer to the `output` matrix.

The `fft_mag` function is equivalent to `rfft_mag` and is provided for compatibility with previous versions of VisualDSP++.

When using the `rfft_mag` function, note that the generated power spectrum will not contain the Nyquist frequency. In cases where the Nyquist frequency is required, the `fft_magnitude` function must be used in conjunction with the `rfft` function.

Algorithm

The algorithm used to calculate the normalized power spectrum is:

$$magnitude(z) = \frac{2\sqrt{Re(a_z)^2 + Im(a_z)^2}}{fftsize}$$

Error Conditions

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

Example

```
#include <filter.h>

#define N 64
float fft_input[N];
complex_float fft_output[N/2];
float spectrum[N/2];
rfft64 (fft_input, fft_output);
rfft_mag (fft_output, spectrum, N);
```

See Also

cfft_mag (SHARC SIMD Processors), fft_magnitude, fftf_magnitude (SHARC SIMD Processors), rfftN (SHARC SIMD Processors)

By default, this function uses SIMD. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

rfftf_2 (SHARC SIMD Processors)

Fast parallel real radix-2 Fast Fourier Transform

Synopsis

```
#include <filter.h>

void rfftf_2 (float data_one_real[], float data_one_imag[],
              float data_two_real[], float data_two_imag[],
              const float twid_real[],
              const float twid_imag[],
              int n);
```

Description

The `rfftf_2` function computes two n-point real radix-2 Fast Fourier Transforms (FFT) using a decimation-in-frequency algorithm. The FFT size `n` must be a power of 2 and not less than 64.

The array `data_one_real` contains the input to the first real FFT, while `data_two_real` contains the input to the second real FFT. Both arrays are expected to be of length `n`. For optimal performance, the arrays should be located in different memory segments. Furthermore, the two input arrays have to be aligned on an address boundary that is a multiple of the FFT size `n`.

The arrays `data_one_imag` and `data_two_imag` of length `n` are used as temporary workspace. At return, they contain the imaginary part of the respective output data set. The arrays should be located in different memory segments.

The size of the twiddle table pointed to by `twid_real` and `twid_imag` must be of size `n/2`. The library function `twidfftf` (on page 2-261) can be used to compute the required twiddle table. The coefficients generated are positive cosine coefficients for the real part and negative sine coefficients for the imaginary part.

> The function invokes the `cfftf` function, which has been implemented to make highly efficient use of the processor's SIMD capabilities and long word addressing mode. The `rfftf_2` function therefore imposes the following restrictions:
>
> - All the arrays that are passed to the function must be allocated in internal memory. The DSP run-time library does not contain a version of the function that can be used with data in external memory.
> - Do not use the function with any application that relies on the `-reserve register[, register...]` switch.
>
> For more information, refer to refer to "Implications of Using SIMD Mode" and "Using Data in External Memory".

Error Conditions

The `rfftf_2` function does not return an error condition.

Example

```
#include <filter.h>

#define FFT_SIZE 64

float dm twidtab_re[FFT_SIZE/2];
float pm twidtab_im[FFT_SIZE/2];

#pragma align 64
float dm fft1_re[FFT_SIZE];
float pm fft1_im[FFT_SIZE];
```

```
#pragma align 64
float dm fft2_re[FFT_SIZE];
float pm fft2_im[FFT_SIZE];

twidfftf (twidtab_re, twidtab_im, FFT_SIZE);

rfftf_2(fft1_re, fft1_im,
        fft2_re, fft2_im,
        twidtab_re, twidtab_im, FFT_SIZE);
```

See Also

cfftf (SHARC SIMD Processors), fftf_magnitude (SHARC SIMD Processors), ifftf (SHARC SIMD Processors), rfft, rfftN (SHARC SIMD Processors). twidfftf (SHARC SIMD Processors)

rfftN

N-point real radix-2 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 *rfft512   (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 `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. 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; for example, `ft8 (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.

If the input data can be overwritten, then the `rfftN` functions allow the array `real_input` to share the same memory as the array `imag_output`. This improves memory usage with only a minimal run-time penalty.

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

These library functions have not been optimized for SHARC SIMD processors. Applications that run on SHARC SIMD processors should use the FFT functions defined in the header file `filter.h`, and described under cfftN (SHARC SIMD Processors) instead.

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];

rfft2048 (real_input, real_output, imag_output);
```

See Also

cfftN, fft_magnitude, ifftN, rfft, rfftN (SHARC SIMD Processors)

The `rfftN` functions make use of circular buffers. Refer to the "Interrupts and Circular Buffering" section of Chapter 1, "Compiler", in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* for information on interrupt dispatcher considerations when circular buffers are used within an application.

rfftN (SHARC SIMD Processors)

N-point real radix-2 Fast Fourier Transform

Synopsis

```
#include <filter.h>

complex_float *rfft65536 (float dm input[],
                          complex_float dm output[]);

complex_float *rfft32768 (float dm input[],
                          complex_float dm output[]);

complex_float *rfft16384 (float dm input[],
                          complex_float dm output[]);

complex_float *rfft8192  (float dm input[],
                          complex_float dm output[]);

complex_float *rfft4096  (float dm input[],
                          complex_float dm output[]);

complex_float *rfft2048  (float dm input[],
                          complex_float dm output[]);

complex_float *rfft1024  (float dm input[],
                          complex_float dm output[]);

complex_float *rfft512   (float dm input[],
                          complex_float dm output[]);

complex_float *rfft256   (float dm input[],
                          complex_float dm output[]);
```

```
complex_float *rfft128   (float dm input[],
                          complex_float dm output[]);

complex_float *rfft64    (float dm input[],
                          complex_float dm output[]);

complex_float *rfft32    (float dm input[],
                          complex_float dm output[]);

complex_float *rfft16    (float dm input[],
                          complex_float dm output[]);
```

Description

The `rfftN` functions are defined in the header file `filter.h`. They have been optimized to take advantage of the SIMD capabilities of the ADSP-211xx, ADSP-212xx, ADSP-213xx , and ADSP-214xx processors. They are therefore not supported by the ADSP-210xx processor family. These FFT functions require complex arguments to ensure that the real and imaginary parts are interleaved in memory and are therefore accessible in a single cycle using the wider data bus of the processor.

Each of these `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 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, or 65536).

There are thirteen distinct functions in this set. 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.

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

```
rfft16 (input, output);
```

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 will use the input array as an intermediate workspace. If the input data is to be preserved, the input array must be first copied to a safe location.

The complex frequency domain signal generated by the rfftN functions is stored in the array output. Because the output signal is symmetric around the midpoint of the frequency domain, the functions only generate N/2 output points.

The rfftN functions do not calculate the Nyquist frequency (which would normally located at output[N/2]). The rfft or cfftN functions should be used in place of these functions if the Nyquist frequency is required.

The rfftN functions return a pointer to the output array.

Error Conditions

The rfftN functions do not return any error conditions.

Example

```
#include <filter.h>

#define N 2048

float input[N];
complex_float output[N/2];

rfft2048 (input, output);
```

See Also

cfftN (SHARC SIMD Processors), ifftN (SHARC SIMD Processors), rfft, rfftN, rfftf_2 (SHARC SIMD Processors)

By default, these functions use SIMD. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

rms

Root mean square

Synopsis

```
#include <stats.h>

float rmsf (const float samples[], int sample_length);
double rms (const double samples[], int sample_length);
long double rmsd (const long double samples[],
                  int sample_length);
```

Description

The root mean square functions return the root mean square of the elements within the input array `samples[]`. The length of the input array is `sample_length`.

Algorithm

The following equation is the basis of the algorithm.

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

where:

a = `samples`

n = `sample_length`

Error Conditions

The root mean square functions do not return an error condition.

Example

```
#include <stats.h>

#define SIZE 256

double data[SIZE];
double result;

result = rms (data, SIZE);
```

See Also

mean, var

For the ADSP-21xxx SIMD processors the `rmsf` function (and `rms`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

rsqrt

Reciprocal square root

Synopsis

```
#include <math.h>

float rsqrtf (float x);
double rsqrt (double x);
long double rsqrtd (long double x);
```

Description

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

Error Conditions

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

Example

```
#include <math.h>

double y;

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

See Also

sqrt

set_flag

Set ADSP-21xxx processor flags

Synopsis

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

Description

The `set_flag` function is used to set the ADSP-21xxx processor 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 `processor_include.h`) or a value [0-3].

Table 2-25. 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 <processor_include.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 <processor_include.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 systems.

- A value of `-1` is returned if the bus is locked and the bus lock time-out is exceeded.
- A value of `0` (zero) is returned if the bus is not locked and a sema-phore is set.

Error Conditions

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

See Also

test_and_set_semaphore

test_and_set_semaphore

Test and set bus lock semaphore

Synopsis

```
#include <processor_include.h>

int test_and_set_semaphore(void *_semaphore,
                           int         _test_value,
                           int         _set_value,
                           int         _timeout);
```

Description

The `test_and_set_semaphore` function is used to control bus lock in multiprocessor ADSP-21xxx systems. The semaphore is only changed if the value of the semaphore is equal to `_test_value`.

The following section lists the return values:

- A value of `-1` is returned if the bus is not locked and a semaphore is set.
- A value of `0` (zero) is returned if the bus is not locked and a semaphore is set.
- A value of `1` is returned if the value of the semaphore is not equal to `_test_value`.

Error Conditions

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

See Also

set_semaphore

timer_off

Disable ADSP-21xxx processor timer

Synopsis

```
#include <processor_include.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 <processor_include.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 ADSP-21065L. Refer to timer0_off, timer1_off (ADSP-21065L Processor Only) to disable the ADSP-21065L programmable timers.

Also, the function is supplied only as an inlined 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 `processor_include.h` header file.

timer0_off, timer1_off (ADSP-21065L Processor Only)

Disable ADSP-21065L processor timers

Synopsis

```
#include <processor_include.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 <processor_include.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 (ADSP-21065L Processor), timer0_off, timer1_off (ADSP-21065L Processor Only)

ⓘ The functions are supplied only as inlined procedures; that is, the compiler substitutes the appropriate statements for any reference to the procedures. Therefore, any source that references `timer0_off` or `timer1_off` must include the `processor_include.h` header file.

timer_on

Enable ADSP-21xxx processor timer

Synopsis

```
#include <processor_include.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 <processor_include.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 ADSP-21065L processor. Refer to “timer0_on, timer1_on (ADSP-21065L Processor)” on page 2-252 to enable the ADSP-21065L programmable timers.

Also, the function is supplied only as an inlined 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 `processor_include.h` header file.

timer_set

Initialize ADSP-21xxx processor timer

Synopsis

```
#include <processor_include.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` registers 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 <processor_include.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 ADSP-21065L. Refer to timer_set to initialize the ADSP-21065L programmable timers.

Also, the function is supplied only as an inlined 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 `processor_include.h` header file.

timer0_on, timer1_on (ADSP-21065L Processor)

Enable ADSP-21065L processor timers

Synopsis

```
#include <processor_include.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 <processor_include.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 (ADSP-21065L Processor Only), timer0_set, timer1_set

The functions are supplied only as inlined 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 `processor_include.h` header file.

timer0_set, timer1_set

Initialize ADSP-21065L processor timers

Synopsis

```
#include <processor_include.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 entry and 50 cycles on return. If `TPERIOD` and `TCOUNT` registers 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 <processor_include.h>
unsigned int hold_tcount;
if (timer0_set (200, 1, 150) != 1)
    timer0_on ();       /* enable timer 0 */
```

See Also

timer0_off, timer1_off (ADSP-21065L Processor Only), timer0_on, timer1_on (ADSP-21065L Processor)

The functions are supplied only as inlined 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 `processor_include.h` header file.

transpm

Matrix transpose

Synopsis

```
#include <matrix.h>

float *transpmf (float dm *output,
                 const float dm *a, int rows, int cols);

double *transpm (double dm *output,
                 const double dm *a, int rows, int cols);

long double *transpmd (long double dm *output,
                       const long double dm *a,
                       int rows, int cols);
```

Description

The `transpm` functions compute the linear algebraic transpose of the input matrix `a[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]`, and `output[cols][rows]`.

The functions return a pointer to the output matrix.

Algorithm

The algorithm for the linear algebraic transpose of a matrix is defined as:

$c_{ji} = a_{ij}$

Error Conditions

The `transpm` functions do not return an error condition.

Example

```
#include <matrix.h>

#define ROWS 4
#define COLS 8

float a[ROWS][COLS];
float a_transpose[COLS][ROWS];

transpmf ((float *)(a_transpose),(float *)(a), ROWS, COLS);
```

See Also

No related function.

The `transpm` functions make use of circular buffers. Refer to the "Interrupts and Circular Buffering" section of Chapter 1, "Compiler", in the *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* for information on interrupt dispatcher considerations when circular buffers are used within an application.

twidfft

Generate FFT twiddle factors

Synopsis

```
#include <filter.h>

complex_float *twidfft(complex_float pm twiddle_tab[],
                       int              fftsize);
```

Description

The `twidfft` function calculates complex twiddle coefficients for an FFT of size `fftsize` and returns the coefficients in the vector `twiddle_tab`. The vector is known as a twiddle table; it contains pairs of cosine and sine values and is used by an FFT function to calculate a Fast Fourier Transform. The table generated by this function may be used by any of the FFT functions `cfft`, `ifft`, and `rfft`. A twiddle table of a given size will contain constant values. Typically, such a table is generated only once during the development cycle of an application and is thereafter preserved by the application in some suitable form.

An application that computes FFTs of different sizes does not require multiple twiddle tables. A single twiddle table can be used to calculate the FFTs, provided that the table is created for the largest FFT that the application expects to generate. Each of the FFT functions `cfft`, `ifft`, and `rfft` have a twiddle stride argument that the application would set to 1 when it is generating an FFT with the largest number of data points. To generate an FFT with half the number of these points, the application would call the FFT functions with the twiddle stride argument set to 2; to generate an FFT with a quarter of the largest number of points, it would set the twiddle stride to 4, and so on.

The function returns a pointer to `twiddle_tab`.

Algorithm

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

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

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

where

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

Error Conditions

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

Example

```
#include <filter.h>

#define N_FFT  128
#define N_FFT2  32

complex_float in1[N_FFT];
complex_float out1[N_FFT];

complex_float in2[N_FFT2];
complex_float out2[N_FFT2];

complex_float temp[N_FFT];
complex_float pm twid_tab[N_FFT / 2];
```

```
twidfft (twid_tab, N_FFT);
cfft (in1, temp, out1, twid_tab, 1, N_FFT);
cfft (in2, temp, out2, twid_tab,
     (N_FFT / N_FFT2) /* twiddle stride 4 */, N_FFT2 );
```

See Also

cfft, ifft, rfft, twidfftf (SHARC SIMD Processors)

twidfftf (SHARC SIMD Processors)

Generate FFT twiddle factors for a fast FFT

Synopsis

```
#include <filter.h>
void twidfftf(float twid_real[], float twid_imag[], int fftsize);
```

Description

The `twidfftf` function generates complex twiddle factors for one of the FFT functions `cfftf`, `ifftf`, or `rfftf_2`. The generated twiddle factors are sets of positive cosine coefficients and negative sine coefficients that the FFT functions will use to calculate the FFT. The function will store the cosine coefficients in the vector `twid_real` and the sine coefficients in the vector `twid_imag`. The size of both the vectors should be `fftsize/2`, where `fftsize` represents the size of the FFT and must be a power of 2 and at least 64.

For maximal efficiency, the `cfftf`, `ifftf`, and `rfftf_2` functions require that the vectors `twid_real` and `twid_imag` are allocated in separate memory blocks.

The twiddle factors that are generated for a specific size of FFT are constant values. Typically, the factors are generated only once during the development cycle of an application and are thereafter preserved by the application in some suitable form.

Algorithm

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

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

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

where

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

Error Conditions

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

Example

```
#include <filter.h>

#define FFT_SIZE 1024

section("seg_dmdata") float twid_r[FFT_SIZE/2];
section("seg_pmdata") float twid_i[FFT_SIZE/2];

#pragma align 1024
section("seg_dmdata") float input_r[FFT_SIZE];
#pragma align 1024
section("seg_pmdata") float input_i[FFT_SIZE];

section("seg_dmdata") float temp_r[FFT_SIZE];
section("seg_pmdata") float temp_i[FFT_SIZE];
```

```
twidfftf(twid_r,twid_i,FFT_SIZE);
cfftf(input_r,input_i,
      temp_r,temp_i,
      twid_r,twid_i,FFT_SIZE);
```

See Also

cfftf (SHARC SIMD Processors), ifftf (SHARC SIMD Processors), rfftf_2 (SHARC SIMD Processors), twidfft

var

Variance

Synopsis

```
#include <stats.h>

float varf (const float a[], int n);
double var (const double a[], int n);
long double vard (const long double a[], int n);
```

Description

The variance functions return the variance of the input array `a[]`. The length of the input array is `n`.

Algorithm

The following equation is the basis of the algorithm.

$$c = \frac{n \sum_{i=0}^{n-1} a_i^2 - \left(\sum_{i=0}^{n-1} a_1 \right)^2}{n(n-1)}$$

Error Conditions

The variance functions do not return an error condition.

Example

```
#include <stats.h>
#define SIZE 256

double data[SIZE];
double result;

result = var (data, SIZE);
```

See Also

mean

For the ADSP-21xxx SIMD processors the `varf` function (and `var`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

vecdot

Vector dot product

Synopsis

```
#include <vector.h>

float vecdotf (const float dm a[],
               const float dm b[], int samples);

double vecdot (const double dm a[],
               const double dm b[], int samples);

long double vecdotd (const long double dm a[],
                     const long double dm b[], int samples);
```

Description

The `vecdot` functions compute the dot product of the vectors `a[]` and `b[]`, which are `samples` in size. They return the scalar result.

Algorithm

The following equation is the basis of the algorithm.

$$return = \sum_{i=0}^{samples-1} a_i \bullet b_i$$

Error Conditions

The `vecdot` functions do not return an error condition.

Example

```
#include <vector.h>

#define N 100

double x[N], y[N];
double answer;

answer = vecdot (x, y, N);
```

See Also

cvecdot

For the ADSP-21xxx SIMD processors the `vecdotf` function (and `vecdot`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to “Implications of Using SIMD Mode” on page 2-23 for more information.

vecsadd

Vector + scalar addition

Synopsis

```
#include <vector.h>

float *vecsaddf (const float dm a[], float scalar,
                 float dm output[], int samples);

double *vecsadd (const double dm a[], double scalar,
                 double dm output[], int samples);

long double *vecsaddd (const long double dm a[],
                       long double scalar,
                       long double dm output[],
                       int samples);
```

Description

The `vecsadd` functions compute the sum of each element of the vector `a[]`, added to the scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The `vecsadd` functions do not return an error condition.

Example

```
#include <vector.h>

#define N 100

double input[N], result[N];
double x;

vecsadd (input, x, result, N);
```

See Also

cvecsadd, vecsmlt, vecssub, vecvadd

For the ADSP-21xxx SIMD processors the `vecsaddf` function (and `vecsadd`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to “Implications of Using SIMD Mode” on page 2-23 for more information.

vecsmlt

Vector * scalar multiplication

Synopsis

```
#include <vector.h>

float *vecsmltf (const float dm a[], float scalar,
                 float dm output[], int samples);

double *vecsmlt (const double dm a[], double scalar,
                 double dm output[], int samples);

long double *vecsmltd (const long double dm a[],
                       long double scalar,
                       long double dm output[],
                       int samples);
```

Description

The vecsmlt functions compute the product of each element of the vector a[], multiplied by the scalar. Both the input and output vectors are samples in size. The functions return a pointer to the output vector.

Error Conditions

The vecsmlt functions do not return an error condition.

Example

```
#include <vector.h>

#define N 100

double input[N], result[N];
double x;

vecsmlt (input, x, result, N);
```

See Also

cvecsmlt, vecsadd, vecssub, vecvmlt

For the ADSP-21xxx SIMD processors the `vecsmltf` function (and `vecsmlt`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

vecssub

Vector – scalar subtraction

Synopsis

```
#include <vector.h>

float *vecssubf (const float dm a[], float scalar,
                 float dm output[], int samples);

double *vecssub (const double dm a[], double scalar,
                 double dm output[], int samples);

long double *vecssubd (const long double dm a[],
                       long double scalar,
                       long double dm output[],
                       int samples);
```

Description

The `vecssub` functions compute the difference of each element of the vector `a[]`, minus the scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The `vecssub` functions do not return an error condition.

Example

```
#include <vector.h>

#define N 100

double input[N], result[N];
double x;

vecssub (input, x, result, N);
```

See Also

cvecssub, vecsadd, vecsmlt, vecvsub

For the ADSP-21xxx SIMD processors the `vecssubf` function (and `vecssub`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

vecvadd

Vector + vector addition

Synopsis

```
#include <vector.h>

float *vecvaddf (const float dm a[], const float dm b[],
                 float dm output[], int samples);

double *vecvadd (const double dm a[], const double dm b[],
                 double dm output[], int samples);

long double *vecvaddd (const long double dm a[],
                       const long double dm b[],
                       long double dm output[],
                       int samples);
```

Description

The `vecvadd` functions compute the sum of each of the elements of the vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The `vecvadd` functions do not return an error condition.

Example

```
#include <vector.h>
#define N 100

double input_1[N];
double input_2[N], result[N];

vecvadd (input_1, input_2, result, N);
```

See Also

cvecvadd, vecsadd, vecvmlt, vecvsub

For the ADSP-21xxx SIMD processors the `vecvaddf` function (and `vecvadd`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

vecvmlt

Vector * vector multiplication

Synopsis

```
#include <vector.h>

float *vecvmltf (const float dm a[], const float dm b[],
                 float dm output[], int samples);

double *vecvmlt (const double dm a[], const double dm b[],
                 double dm output[], int samples);

long double *vecvmltd (const long double dm a[],
                       const long double dm b[],
                       long double dm output[],
                       int samples);
```

Description

The `vecvmlt` functions compute the product of each of the elements of the vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The `vecvmlt` functions do not return an error condition.

Example

```
#include <vector.h>

#define N 100

double input_1[N];
double input_2[N], result[N];

vecvmlt (input_1, input_2, result, N);
```

See Also

cvecvmlt, vecsmlt, vecvadd, vecvsub

For the ADSP-21xxx SIMD processors the `vecvmltf` function (and `vecvmlt`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

vecvsub

Vector – vector subtraction

Synopsis

```
#include <vector.h>

float *vecvsubf (const float dm a[], const float dm b[],
                float dm output[], int samples);

double *vecvsub (const double dm a[], const double dm b[],
                double dm output[], int samples);

long double *vecvsubd (const long double dm a[],
                      const long double dm b[],
                      long double dm output[],
                      int samples);
```

Description

The `vecvsub` functions compute the difference of each of the elements of the vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The `vecvsub` functions do not return an error condition.

Example

```
#include <vector.h>

#define N 100

double input_1[N];
double input_2[N], result[N];

vecvsub (input_1, input_2, result, N);
```

See Also

cvecvsub, vecssub, vecvadd, vecvmlt

For the ADSP-21xxx SIMD processors the `vecvsubf` function (and `vecvsub`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to "Implications of Using SIMD Mode" on page 2-23 for more information.

zero_cross

Count zero crossings

Synopsis

```
#include <stats.h>

int zero_crossf (const float in[], int length);
int zero_cross (const double in[], int length);
int zero_crossd (const long double in[], int length);
```

Description

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

Error Conditions

The `zero_cross` functions do not return an error condition.

Example

```
#include <stats.h>

#define SIZE 256

double input[SIZE];
int result;

result = zero_cross (input, SIZE);
```

See Also

No related function.

I INDEX

D

F

G

H

L

M

U

V