CrossCore® Embedded Studio

CCES 2.9.0 C/C++ Library Manual for SHARC Processors

(Includes SHARC+ and ARM Processors)

Revision 2.2, May 2019

Part Number
82-100118-01

Analog Devices, Inc.
One Technology Way
Norwood, MA 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, Blackfin+, CrossCore, EngineerZone, EZ-Board, EZ-KIT, EZ-KIT Lite, EZ-Extender, SHARC, SHARC+, and VisualDSP++ are registered trademarks of Analog Devices, Inc.

EZ-KIT Mini is a trademark of Analog Devices, Inc.

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

Contents

DSP Run-Time Library

1 Preface

Thank you for purchasing CrossCore® Embedded Studio (CCES), Analog Devices development software for SHARC® and SHARC+® digital-signal processors.

Purpose of This Manual

The *C/C++ Library Manual for SHARC Processors* contains information about the C/C++ and DSP run-time libraries for SHARC (ADSP-21xxx) and SHARC+ (ADSP-SCxxx) 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 are programmers who are familiar with Analog Devices SHARC processors. This manual assumes that the audience has a working knowledge of the SHARC processors' architecture and instruction set and C/C++ programming languages.

Programmers who are unfamiliar with SHARC processors can use this manual, but should supplement it with other texts (such as the appropriate hardware reference, programming reference, and data sheet) that provide information about their SHARC processor architecture and instructions).

Manual Contents

The manual consists of:

- 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).

Technical Support

You can reach Analog Devices processors and DSP technical support in the following ways:

- Post your questions in the processors and DSP support community at EngineerZone®:

  http://ez.analog.com/community/dsp

- Submit your questions to technical support directly at:

  http://www.analog.com/support

- E-mail your questions about processors, DSPs, and tools development software from ***CrossCore Embedded Studio*** or ***VisualDSP++***®:

  Choose ***Help*** > ***Email Support***. This creates an e-mail to processor.tools.support@analog.com and automatically attaches your CrossCore Embedded Studio or VisualDSP++ version information and `license.dat` file.

- E-mail your questions about processors and processor applications to:

  processor.tools.support@analog.com

  processor.china@analog.com

- Contact your Analog Devices sales office or authorized distributor. Locate one at:

  http://www.analog.com/adi-sales

- Send questions by mail to:

```
Analog Devices, Inc.
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
USA
```

Supported Processors

The names *SHARC* and *SHARC+* refer 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. Refer to the CCES online help for a complete list of supported processors.

Product Information

Product information can be obtained from the Analog Devices website and CrossCore Embedded Studio online help system.

Analog Devices Website

The Analog Devices website, http://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 website 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.

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.

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*
File > Close	Titles in bold style indicate the location of an item within the CrossCore Embedded Studio IDE'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 ellipsis; read the example as an optional comma-separated list of `this`.

Example	*Description*
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with `letter gothic` font and italic style format.
NOTE:	*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:	*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.
ATTENTION:	*ATTENTION:* 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 *ATTENTION:* appears instead of this symbol.

2 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 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.
- Documented Library Functions 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. The DSP Run-Time Library chapter describes a number of signal processing, matrix, and statistical functions that assist code development.

NOTE: 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 in the CCES online help.

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
- Linking Library Functions
- Library Attributes
- Working With Library Header Files
- Calling Library Functions From an ISR
- Using the Libraries in a Multi-Threaded Environment
- Using Compiler Built-In C Library Functions
- Abridged C++ Library Support
- Measuring Cycle Counts
- File I/O Support
- Fatal Error Handling

For information on the C library's contents, see C and C++ Run-Time Libraries Guide. For information on the Abridged C++ library's contents, see Abridged C++ Library Support.

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 and C++ Run-Time Libraries Guide.

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.

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 the *C/C++ Compiler Manual for SHARC Processors*.

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

Linking Library Functions

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 one of the provided Linker Description files (either generated or non-generated). Linker Description files have the `.ldf` extension.

If you are using a customized `.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 *Linker and Utilities Manual.*

Functional Breakdown

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

- Compiler support library - Contains internal functions that support the in-line code generated by the compiler; emulated arithmetic is a typical case.
- C run-time library - Comprises all the functions that are defined by the ANSI standard, plus various Analog Devices extensions.
- DSP run-time library - Contains additional library functions supplied by Analog Devices that provide services commonly required by DSP applications.
- Heap debugging library - Contains debug versions of the heap support provided by the C/C++ run-time library, as well as some additional diagnostic functions relating to heap use.
- Instrumented profiling library - Contains support routines for recording the cycles spent in each profiled function.
- I/O library - Supports a subset of the C standard's I/O functionality.
- Dynamic module loader library (`libldr`)- Supports loading and using dynamically-loadable modules created using the `elfloader` utility.
- Dynamic module loader library (`libdyn`)- Supports loading and using dynamically-loadable modules created using the `elf2dyn` utility. This support is deprecated and will be removed in a future release. The support provided by the `libldr` library should be used instead.

In addition to regular run-time libraries, CCES has some additional libraries which provide variants of `LibIO` (the I/O run-time support library). These variants are:

- `libio*_lite.dlb` - libraries which provide smaller versions of `LibIO` with more limited functionality. These smaller `LibIO` libraries can be used by specifying the following switch on the build command line: `-flags-link -MD__LIBIO_LITE`.
- `libio*_fx.dlb` - libraries which provide versions of `LibIO` with full support for the fixed-point format specifiers for the `fract` types. These libraries can be used by specifying the following switch on the build command line: `-flags-link -MD__LIBIO_FX`.

Library Location

The C/C++ run-time libraries are provided in binary form in the `SHARC\lib\`*`processor`*`_rev_`*`revision`* directories, where:

- `processor` identifies which processor for which the library is built, and is the processor's name with the leading `ADSP-` stripped.
- `revision` identifies for which silicon revision the library is built. For example, a revision of 0.1 indicates that the library is built with the command-line switch `-si-revision 0.1`.

So the directory `SHARC\lib\21266_rev_any` contains libraries that have been built with `-proc ADSP-21266 -si-revision any` switches.

The C/C++ run-time libraries are provided in source form, where available, in the directories named `SHARC\lib\src\libname`, where `libname` indicates which library the source is used to build.

Library Selection

The library directory used to link an application is selected through the `-proc` and `-si-revision` compiler switches, in conjunction with an XML configuration file.

The `-proc` switch directs the compiler driver to read an XML configuration file from `System\ArchDef`, based on the selected processor. For example, a compiler switch of `-proc ADSP-21266` would cause the compiler driver to read the `ADSP-21266-compiler.xml` file in `System\ArchDef`.

Each such XML file indicates which library subdirectory should be used, for supported silicon revision of that processor. For example, the XML file for the ADSP-21266 processor indicates that for silicon revision 0.2, the library directory to use is `sharc\lib\21266_rev_any`.

A given library subdirectory might support more than one silicon revision. In such cases, the XML file will give the same library subdirectory for several silicon revisions.

Library Naming

Within the library subdirectories, the libraries follow a consistent naming scheme, so that the library's name will be `lib<name><attrs>.dlb`, where name indicates the library's purpose, and *attrs* is a sequence of zero or more attributes. The ***C/C++ Library Names*** table shows the library's names, and the ***Library Name Attributes*** table shows the enumerated attributes.

Table 2-1: C/C++ Library Names

Description	Library Name	Comments
Compiler support library	libcc*.dlb	
C run-time library	libc*.dlb	
C++ run-time library	libcpp*.dlb	
DSP run-time library	libdsp*.dlb	
Heap debugging library	libheapdbg*.dlb	
Instrumented profiling library	libprofile*.dlb	
I/O run-time library	libio*.dlb	

Table 2-1: C/C++ Library Names (Continued)

Description	Library Name	Comments
I/O run-time library with no support for alternative device drivers or printf("%a")	libio_lite*.dlb	
I/O run-time library with full support for the fixed-point format specifiers	libiofx*.dlb	
Loader library for dynamically-loadable modules (DLMs)	libldr*.dlb	Operates on DLMs in loader stream form. See Dynamically-Loadable Modules in the System Runtime documentation.
Deprecated loader library for dynamically-loadable modules (DLMs)	libdyn*.dlb	Deprecated library that operates on DLMs produced by elf2dyn.

Table 2-2: Library Name Attributes

Attribute	Meaning
mt	Built with -threads, for use in a multi-threaded environment
x	Built with -eh -rtti, to enable C++ exception-handling

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.

The libraries for the ADSP-214xx processor are built in short-word mode.

Library Startup Files

The library subdirectories also contain object files which contain the "run-time header", or "C run-time" (CRT) startup code. These files contain the code that is executed when your application first starts running; it is this code that configures the expected C/C++ environment and passes control to your main() function.

Startup files have names of the form *procid_attrs*_hdr.doj:

- *procid* indicates which processor the startup code is for; for ADSP-211xx, ADSP-212xx, and ADSP-213xx processors, this is the last three digits of the processor's name. For other processors, this is the five digits of the processor's name.
- *attrs* is a list of zero or more names indicating which features are configured by the startup code. These attributes and their meanings are listed in the ***Startup File Attributes*** table.

Table 2-3: Startup File Attributes

Attribute	*Meaning*
_cpp	C++ startup file
_sov	Enables stack overflow detection

Library Attributes

The run-time libraries use file attributes. See the *C/C++ Compiler Manual for SHARC Processors* for details on how to use file attributes. Each library function has a defined set of file attributes that are listed in the ***Run-Time Library Object Attributes*** table. For each object `obj` in the run-time libraries, the following is true.

Table 2-4: 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`. For example, suppose that `obj` were part of `libdsp.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`.
`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);
```

File: `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 the ***Attribute Values in libfunc.dlb*** table.

Table 2-5: 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`

Table 2-5: Attribute Values in libfunc.dlb (Continued)

Attribute	*Value*
subfunc.doj libGroup libGroup libName libFunc libFunc libFunc libFunc libFunc libFunc FuncName prefersMem prefersMemNum	func1.h func2.h(*3) libfunc _func1 func1 _func2 func2 _subfunc subfunc _subfunc internal(*4) 30

*1 func1.doj is not be performance critical, based on its normal usage.

*2 func2.doj is 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 functions 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.

The ***Additional libGroup Attribute Values*** table lists additional libGroup attribute values.

Table 2-6: Additional libGroup Attribute Values

Value	*Meaning*
floating_point_support	Compiler support routines for floating-point arithmetic
fixed_point_support	Compiler support routines for native fixed-point types
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
runtime_checking	Run-time checks to provide support for dynamic checks

Table 2-6: Additional libGroup Attribute Values (Continued)

`stack_overflow_detection`	Run-time checks to support detection of stack overflow
`libprofile`	Run-time functions to support profiling

Objects with any of the `libGroup` attribute values listed in the ***Additional libGroup Attribute Values*** table will not contain any `libGroup` or `libFunc` attributes from any calling objects.

The ***Default Memory Placement Summary*** table presents a summary of the default memory placement using `prefersMem`.

Table 2-7: Default Memory Placement Summary

Library	*Placement*
`libcpp*.dlb`	`any`
`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 *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 the ***Standard C Run-Time Library Header Files*** table. You should use a C standard text to augment the information supplied in this chapter.

Table 2-8: Standard C Run-Time Library Header Files

Header	*Purpose*	*Standard*
adi_libldr.h	Dynamically-loadable modules (adi_libldr.h)	Analog Extension
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
errno.h	Error handling	ANSI
float.h	Floating point	ANSI
heap_debug.h	Macros and prototypes for heap debugging	Analog Extension
instrprof.h	Instrumented profiling support (instrprof.h)	Analog Extension
iso646.h	Boolean operators	ANSI
libdyn.h	Deprecated support for dynamically-loadable modules (libdyn.h)	Analog Extension
limits.h	Limits	ANSI
locale.h	Localization	ANSI
math.h	Mathematics	ANSI
misra_types.h	Exact-width integer types	MISRA-C:2004
pgo_hw.h	Profile-guided optimization support (pgo_hw.h)	Analog Extension
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
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_libldr.h

The adi_libldr.h header file contains type definitions and function declarations for loading dynamically-loadable modules (DLMs) that have been converted into loader streams. adi_libldr.h replaces the deprecated support for DLMs provided by the libdyn.h header. See ***Dynamically-Loadable Modules*** in the ***System Runtime Documentation*** section of help for information on how to use the adi_libldr.h header.

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. For more information about using the pragma, see ***Linking Control Pragmas*** in the *C/C++ Compiler Manual for SHARC Processors*.

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 IDE: choose ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Compiler*** > ***Additional Options***.

ctype.h

The ctype.h header file contains functions for character handling, such as isalpha, tolower, etc.

For a list of library functions that use this header, see the ***Library Functions in the ctype.h Header File*** table in Documented Library Functions.

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.

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.

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).

heap_debug.h

The `heap_debug.h` header file defines a set of functions and macros for configuring and manipulating the heap debugging library.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

When the macro _HEAP_DEBUG is not defined, the functions defined in heap_debug.h are replaced by simple statements representing a successful return from that function. This ensures that any code using these functions will link and operate as expected without any performance degradation when heap debugging is disabled.

Configuration macros are provided in this file, which represent the values of the bit-fields used to control the behavior of the heap debugging. These configuration macros are shown in the ***Configuration Macros for Heap Debugging*** table.

Table 2-9: Configuration Macros for Heap Debugging

Macro	*Use*
_HEAP_STDERR_DIAG	Enable/disable diagnostics about heap usage via stderr
_HEAP_HPL_GEN	Enable/disable generation of the .hpl file used for heap debugging report
_HEAP_TRACK_USAGE	Enable/disable tracking of heap usage

These macros can be used as parameters to adi_heap_debug_enable and adi_heap_debug_disable to enable or disable features at runtime. Tracking of heap usage is implicitly enabled when either report generation or run-time diagnostics are enabled at runtime. For more information, see ***Enabling And Disabling Features*** in the *C/C++ Compiler Manual for SHARC Processors*.

Macros representing various categories of heap error are defined in heap_debug.h. These values can be used as parameters to the adi_heap_debug_set_error, adi_heap_debug_set_ignore, and adi_heap_debug_set_warning functions at runtime, or as definitions for the "C" unsigned long variables adi_heap_debug_error, __heap_debug_ignore and __heap_debug_warning at build-time in order to configure the severity of these error types when runtime diagnostics are enabled. These error type macros are shown in the ***Error Type Macros for Heap Debugging*** table. For more information on using these macros, see ***Setting the Severity Of Error Messages*** in the *C/C++ Compiler Manual for SHARC Processors*.

Table 2-10: Error Type Macros for Heap Debugging

Macro	*Error*
_HEAP_ERROR_UNKNOWN	An unknown error has occurred
_HEAP_ERROR_FAILED	An allocation has been unsuccessful
_HEAP_ERROR_ALLOCATION_OF_ZERO	An allocation has been requested with size of zero
_HEAP_ERROR_NULL_PTR	A null pointer has been passed where not expected
_HEAP_ERROR_INVALID_ADDRESS	A pointer has been passed which doesn't correspond to a block on the heap
_HEAP_ERROR_BLOCK_IS_CORRUPT	Corruption has been detected in the heap
_HEAP_ERROR_FREE_OF_FREE	A deallocation of an already de-allocated block has been requested
_HEAP_ERROR_FUNCTION_MISMATCH	An unexpected function is being used to de-allocate a block (i.e. calling free on an block allocated by new)
_HEAP_ERROR_UNFREED_BLOCK	A memory leak has been detected
_HEAP_ERROR_WRONG_HEAP	A heap operation has the wrong heap index specified

Table 2-10: Error Type Macros for Heap Debugging (Continued)

Macro	*Error*
_HEAP_ERROR_INVALID_INPUT	An invalid parameter has been passed to a heap debugging function
_HEAP_ERROR_INTERNAL	An internal error has occurred
_HEAP_ERROR_IN_ISR	The heap has been used within an interrupt
_HEAP_ERROR_MISSING_OUTPUT	A block was deleted, freed or reallocated with a different address mode (-char-size-8/32) to where it was allocated.
_HEAP_ERROR_ADDRESSING_MISMATCH	Report output has been lost due to insufficient or no buffer space
_HEAP_ERROR_ALL	Refers to all of the above errors collectively

instrprof.h

The `instrprof.h` header file declares user-callable functions in support of instrumented profiling. For more information, see ***Profiling With Instrumented Code*** in the *C/C++ Compiler Manual for SHARC Processors*.

iso646.h

The `iso646.h` header file defines symbolic names for certain C operators; the symbolic names and their associated value are shown in the ***Symbolic Names Defined in iso646.h*** table.

Table 2-11: Symbolic Names Defined in iso646.h

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

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

libdyn.h

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. See ***Dynamically-Loadable Modules*** in the ***System Runtime Documentation*** section of help for information on how to use the APIs in adi_libldr.h to load a DLM from a loader stream.

The `libdyn.h` header file contains type definitions and function declarations for loading DLMs that have been produced by the `elf2dyn` utility. For more information on `elf2dyn`, refer to the ***Loader and Utilities Manual***. For information on how to create and use DLMs, refer to the *System Run-Time Documentation* in the online help.

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 the ***Library Functions in the locale.h Header File*** table in Documented Library Functions.

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 the ***Library Functions in the math.h Header File*** table in Documented Library Functions.

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 the DSP Run-Time Library chapter 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 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 and stdbool.h, plus data types `char_t`, `float32_t`, and `float64_t`.

pgo_hw.h

The `pgo_hw.h` header file declares user-callable functions in support of profile-guided optimization, when used with hardware rather than a simulator. For more information, see ***Profile Guided Optimization and Code Coverage*** in the *C/C++ Compiler Manual for SHARC Processors*.

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 the ***Library Functions in the setjmp.h Header File*** table in Documented Library Functions.

signal.h

The `signal.h` header file provides function prototypes for the standard ANSI `signal.h` routines.

For a list of library functions that use this header, see the ***Library Functions in the signal.h Header File*** table in Documented Library Functions.

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 the ***Library Functions in the stdarg.h Header File*** table in Documented Library Functions.

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 *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.

The ***Exact-Width Integer Types*** table describes each type with regard to `MIN` and `MAX` macros.

Table 2-12: 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 2-12: Exact-Width Integer Types (Continued)

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`

The ***MIN and MAX Macros for typedefs in Other Headings*** table describes `MIN` and `MAX` macros defined for typedefs in other headings.

Table 2-13: 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). 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.

The implementation of both I/O libraries is based on a simple interface provided by the CCES simulator and processor evaluation systems; for further details of this interface, refer to the System Run-Time Documentation in the online help.

The following restrictions apply to this software release:

- The functions tmpfile and tmpnam are not available
- The functions rename and remove are always delegated to the current default device driver
- 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.

NOTE: 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 the ***Library Functions in the stdio.h Header File*** table in Documented Library Functions.

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`. The ***Standard Library - Additional Functions*** table is a summary of the additional library functions defined by the `stdlib.h` header file.

> **NOTE:** Some functions exist as both integer and floating point functions. The floating point functions typically have an `f` prefix. Ensure you use the correct type.

Table 2-14: 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);`
	`int heap_init(int index);`
	`int heap_install(void *base, size_t size, int userid);`
	`int heap_lookup(int userid);`
	`void *heap_malloc(int heap_index, size_t size);`
	`void *heap_realloc(int heap_index, void *ptr, size_t size);`
	`int heap_space_unused(int index);`
	`int heap_switch(int heapid);`
	`int space_unused(void);`

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.

NOTE: 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 the ***Library Functions in the stdlib.h Header File*** table in Documented Library Functions.

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 the ***Library Functions in the string.h Header File*** table in Documented Library Functions.

sys/adi_core.h

The `sys/adi_core.h` header file declares functions and enumerations that are used for multi-core processors (see the ***Library Functions in the sys/adi_core.h Header File*** table in Documented Library Functions.

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: `time_t` and `clock_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             */
};
```

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 compilation time (either before including the header file `time.h`, or by using the compilation time switch `-D__LONG_LONG_PROCESSOR_TIME__`), the `clock_t` data type is `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.

NOTE: 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*> compilation time switch
- Via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Compiler*** > ***Processor*** > ***Processor speed (MHz)***
- From the header file cycles.h

For a list of library functions that use this header, see the ***Library Functions in the time.h Header File*** table in Documented Library Functions.

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
- 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.

The memory allocation routines malloc, calloc, realloc, free, the C++ operators new and delete, and any variants, read and update global tables and are not interrupt-safe. The heap debugging library can detect calls to memory allocation routines from an ISR, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

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 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 an RTOS, 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 Calling Library Functions From an ISR.

A CCES 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`).

Note that the DSP run-time library (which is described in 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 built-in functions (sometimes called *intrinsics*) 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 the `builtins.h` header file. Include this header file in your program to use these functions.

Typically, inline built-in functions are faster than an average library routine, as they do not incur the calling overhead. The routines in the ***Compiler Built-in Functions*** table are built-in C library functions for the `cc21k` compiler.

Table 2-15: Compiler Built-in Functions

abs	avg	clip
copysign*1	copysignf	fabs1
fabsf	favg1	favgf
fclip1	fclipf	fmax1
fmaxf	fmin1	fminf
labs	lavg	lclip
lmax	lmin	max
memcpy*2	memmove2	min
strcpy2	strlen2	

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

*2 Not all references to these functions are inlined. Some generate a call to a library function if the compiler does not have sufficient information about the arguments to generate efficient inline code.

If you want to use the C run-time library functions of the same name instead of the built-in function, refer to ***builtins.h*** in the *C/C++ Compiler Manual for SHARC Processors*.

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.

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 in the CCES online help.

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

- Embedded C++ Library Header Files
- C++ Header Files for C Library Facilities
- Embedded Standard Template Library Header Files
- Header Files for C++ Library Compatibility

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.

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.

NOTE: 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 is `foo.h`, the name of the equivalent C++ header file is `cfoo`. For example, the C++ header file `<cstdio>` provides the same facilities as the C header file `<stdio.h>`.

The ***C++ Header Files for C Library Facilities*** table 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 2-16: 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

Table 2-16: C++ Header Files for C Library Facilities (Continued)

Header	*Description*
`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

NOTE: The DSP Run-Time Library chapter 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. The ***Header Files for C++ Library Compatibility*** table describes these files.

Table 2-17: 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

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
- Cycle Counting Facility With Statistics
- Using time.h to Measure Cycle Counts

- Determining the Processor Clock Rate
- Considerations When Measuring Cycle Counts

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 that the header file `cycle_count.h` defines as:

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

NOTE: 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 compilation 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 and Using time.h to Measure Cycle Counts); the relative benefits of this facility are outlined in Considerations When Measuring Cycle Counts.

The basic cycle counting facility is based upon macros. Therefore, it can 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 compilation 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);

      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 and Using time.h to Measure Cycle Counts. Also refer to Considerations When Measuring Cycle Counts, which provides some useful tips with regards to performance measurements.

Using time.h to Measure Cycle Counts

The `time.h` header file defines the `clock_t` data type, `clock` function, and `CLOCKS_PER_SEC` macro, which together can 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).
- 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 and Cycle Counting Facility With Statistics, respectively. Also refer to Considerations When Measuring Cycle Counts 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 the compilation time switch, `-DCLOCKS_PER_SEC=<definition>`
- By the ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Compiler*** > ***Processor*** > ***Processor speed (MHz)*** IDE option
- 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 usually is 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. 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. 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` compilation 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. 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 compilation time macro `__LONG_LONG_PROCESSOR_TIME__`.

File I/O Support

The CCES 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, among others. The functions defined in the `stdio.h` header file make use of these primitives to provide conventional C input and output facilities. For details on File I/O support, refer to the system run-time documentation.

Refer to stdio.h for information about the conventional C input and output facilities that are provided by the compiler.

Fatal Error Handling

The CCES run-time library provides a global mechanism for handling non-recoverable (fatal) errors, encountered during application execution. This is provided by the adi_fatal_error and adi_fatal_exception functions, which write information related to the encountered error before looping around the `__fatal_error` and `__fatal_exception` breakpoints.

These items of information can be stored regarding the encountered error:

- General code indicating the source of the error
- Specific code indicating the actual error that occurred
- A PC address indicating where the error is reported
- A value related to the error. This may not be relevant and may be left empty.

This information is stored in global variables, detailed in the ***Global Variables Used In Fatal Error Reporting*** table. Each variable is 32 bits in size. The value related to the error can be interpreted in different ways, depending on the error with which it is associated.

Table 2-18: Global Variables Used In Fatal Error Reporting

Use	*Label*	*Type*
General code	`_adi_fatal_error_general_code`	Integer
Specific code	`_adi_fatal_error_specific_code`	Integer
PC	`_adi_fatal_error_pc`	Memory address
Value	`_adi_fatal_error_value`	Depends on error

FatalError.xml

The `FatalError.xml` file, contained in the `System` directory of the CCES installation, details the relationships between general codes and specific codes, and provides additional detail on the specific code, such as a description of the error.

A general code is associated with a list of specific codes, though a list of specific codes can be associated with one or more general codes. Specific code values must be unique within a list of specific codes, but duplicate specific codes are allowed if they are within separate lists.

General Codes

`LibraryError` is a general code associated with the run-time libraries. It refers to errors identified with the run-time libraries. An additional general code, `UserError`, is available for any user-defined error values. The values representing these codes are shown in the ***General Error Codes Used By Run-Time Library*** table.

Table 2-19: General Error Codes Used By Run-Time Library

General Code	*Name*	*Value*
Library error	`LibraryError`	`0x7`
Run-time error	`RunTimeError`	`0x8`
Errno values	`Errno`	`0xB`
User defined error	`UserError`	`0xffffffff`

Library Error-Specific Codes

Specific codes, associated with the `LibraryError` general code, provide details for any fatal errors that can be identified by the run-time libraries. These errors are described in the ***Library Error Specific Codes*** table.

Table 2-20: Library Error Specific Codes

Specific Code Value	*Error*	*Description*	*Error Value Interpretation*
`0x2`	`InsufficientHeapForLibrary`	An allocation from the default heap in the system libraries has failed.	None
`0x3`	`IONotAllowed`	I/O has been requested when scheduling has been disabled, or from within an ISR.	None
`0x4`	`ProfBadExeName`	Profiling/heap debugging has failed due to an invalid application filename.	None
`0x5`	`OSAL BindingError`	An operating system abstraction layer function has failed.	None
`0x6`	`adi_osal_Init_failure`	The call to `adi_osal_Init` made from the CRT startup code returned an error.	None
`0x101`	`HeapUnknown`	An unknown heap debugging error has occurred.	None
`0x102`	`HeapFailed`	A heap operation has failed.	None
`0x103`	`HeapAllocationOfZero`	A heap allocation of zero has been detected.	None
`0x104`	`HeapNullPointer`	A heap operation using an unexpected null pointer has been detected.	None
`0x105`	`HeapInvalidAddress`	A heap operation using an invalid address has been detected.	Pointer to invalid address
`0x106`	`HeapBlockIsCorrupt`	A corrupt block has been detected on the heap.	Pointer to corrupt block
`0x107`	`HeapReallocOfZero`	A call to realloc with no pointer or size has been detected.	None

Table 2-20: Library Error Specific Codes (Continued)

Specific Code Value	*Error*	*Description*	*Error Value Interpretation*
`0x108`	`HeapFunctionMisMatch`	A heap operation incompatible with the block being manipulated has been detected.	Pointer to block being manipulated
`0x109`	`HeapUnfreedBlock`	An unfreed block on the heap has been detected.	Pointer to unfreed block
`0x10a`	`HeapWrongHeap`	A heap operation using the wrong heap has been detected.	Pointer to block being manipulated
`0x10b`	`HeapAllocationTooLarge`	A heap allocation request larger than the heap that it is being allocated to has been detected.	None
`0x10c`	`HeapInvalidInput`	A heap operation has been given an invalid input.	None
`0x10d`	`HeapInternalError`	An internal error has occurred within the heap debugging library.	None
`0x10e`	`HeapInInterrupt`	The heap has been used within an interrupt.	None
`0x10f`	`HeapMissingOutput`	There is output missing from the heap report file, due to insufficient buffering.	Unsigned integer counting number of missing 8-bit bytes
`0x110`	`HeapInsufficientSpace`	Heap debugging has failed, due to insufficient available heap space.	None
`0x111`	`HeapCantOpenDump`	Heap debugging cannot open heap dump file.	None
`0x112`	`HeapCantOpenTrace`	Heap debugging cannot open `.hpl` file for report output.	None
`0x113`	`HeapInvalidHeapID`	An invalid heap id has been used.	ID of invalid heap
`0x114`	`HeapAddressingMismatch`	A block was deleted, freed or reallocated with a different address mode (-char-size-8/32) to where it was allocated.	None
`0x201`	`InstrprofIOFail`	Instrumented profiling cannot open its output file.	None
`0x301`	`PGOHWFailedOutput`	The PGO on hardware run-time support failed to open an output file.	None
`0x302`	`PGOHWDataCorrupted`	An internal error has occurred in the PGO on hardware run-time support.	None
`0x303`	`PGOHWInvalidPGO`	The existing PGO data file appears to be corrupted.	None

Errno Values

Specific codes for the `Errno` general code map directly onto the `errno` variable itself. Refer to `errno.h` for interpretation of the values.

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.

The ***Library Functions in the adi_libldr.h Header File*** table lists the library functions in the `adi_libldr.h` header file. Refer to adi_libldr.h for more information on this header file.

Table 2-21: Library Functions in the adi_libldr.h Header File

adi_libldr_Load	adi_libldr_LoadDbg	adi_libldr_LookUpSym

The ***complex.h Library Functions*** table lists functions in the `complex.h` header file. For more information, see complex.h.

Table 2-22: complex.h Library Functions that operate on C99 complex types

cabs	carg	cexp
cimag	conj	creal

The ***Library Functions in the ctype.h Header File*** table lists the library functions in the `ctype.h` header file. Refer to ctype.h for more information on this header file.

Table 2-23: Library Functions in the ctype.h Header File

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

The ***Library Functions in the heap_debug.h Header File*** table lists the library functions in the `heap_debug.h` header file. Refer to heap_debug.h for more information on this header file.

Table 2-24: Library Functions in the heap_debug.h Header File

adi_dump_all_heaps	adi_dump_heap

Table 2-24: Library Functions in the heap_debug.h Header File (Continued)

adi_heap_debug_disable	adi_heap_debug_enable
adi_heap_debug_end	adi_heap_debug_flush
adi_heap_debug_pause	adi_heap_debug_reset_guard_region
adi_heap_debug_resume	adi_heap_debug_set_buffer
adi_heap_debug_set_call_stack_depth	adi_heap_debug_set_error
adi_heap_debug_set_guard_region	adi_heap_debug_set_ignore
adi_heap_debug_set_warning	adi_verify_all_heaps
adi_verify_heap	

The ***Library Functions in the libdyn.h Header File*** table lists functions in the `libdyn.h` header file. For more information, see libdyn.h.

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Table 2-25: Library Functions in the libdyn.h Header File

dyn_AddHeap	dyn_alloc	dyn_AllocSectionMem
dyn_AllocSectionMemHeap	dyn_CopySectionContents	dyn_FreeEntryPointArray
dyn_FreeSectionMem	dyn_GetEntryPointArray	dyn_GetExpSymTab
dyn_GetHeapForWidth	dyn_GetNumSections	dyn_GetSections
dyn_GetStringTable	dyn_GetStringTableSize	dyn_heap_init
dyn_LookupByName	dyn_RecordRelocOutOfRange	dyn_Relocate
dyn_RetrieveRelocOutOfRange	dyn_RewriteImageToFile	dyn_SetSectionAddr
dyn_SetSectionMem	dyn_ValidateImage	

The ***Library Functions in the locale.h Header File*** table lists the library functions in the `locale.h` header file. Refer to locale.h for more information on this header file.

Table 2-26: Library Functions in the locale.h Header File

localeconv	setlocale

The ***Library Functions in the math.h Header File*** table lists the library functions in the `math.h` header file. Refer to math.h for more information on this header file.

Table 2-27: Library Functions in the math.h Header File

	asin	atan

Table 2-27: Library Functions in the math.h Header File (Continued)

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

The ***Library Functions in the setjmp.h Header File*** table lists the library functions in the `setjmp.h` header file. Refer to setjmp.h for more information on this header file.

Table 2-28: Library Functions in the setjmp.h Header File

longjmp	setjmp

The ***Library Functions in the signal.h Header File*** table lists the library functions in the `signal.h` header file. Refer to signal.h for more information on this header file.

Table 2-29: Library Functions in the signal.h Header File

raise	signal

The ***Library Functions in the stdarg.h Header File*** table lists the library functions in the `stdarg.h` header file. Refer to stdarg.h for more information on this header file.

Table 2-30: Library Functions in the stdarg.h Header File

va_arg	va_end	va_start

The ***Library Functions in the stdfix.h Header File*** table lists the library functions in the `stdfix.h` header file. Refer to stdfix.h for more information on this header file.

Table 2-31: Library Functions in the stdfix.h Header File

absfx	bitsfx	countlsfx
divifx	fxbits	fxdivi
idivfx	mulifx	roundfx
strtofxfx		

The ***Library Functions in the stdio.h Header File*** table lists the library functions in the `stdio.h` header file. Refer to stdio.h for more information on this header file.

Table 2-32: Library Functions in the stdio.h Header File

clearerr	fclose	feof
ferror	fflush	fgetc
fgetpos	fgets	fileno
fopen	fprintf	fputc
fputc	fread	freopen
fscanf	fseek	fsetpos
ftell	fwrite	getc
getchar	gets	ioctl
perror	printf	putc
putchar	puts	remove
rename	rewind	scanf
setbuf	setvbuf	snprintf
sprintf	sscanf	ungetc
vfprintf	vprintf	vsnprintf
vsprintf		

The ***Library Functions in the stdlib.h Header File*** table lists the library functions in the `stdlib.h` header file. Refer to stdlib.h for more information on this header file.

Table 2-33: Library Functions in the stdlib.h Header File

abort	abs	adi_fatal_error
adi_fatal_exception	atexit	atof
atoi	atol	atold
atoll	avg	bsearch
calloc	clip	count_ones
div	exit	free
getenv	heap_calloc	heap_free
heap_init	heap_install	heap_lookup
heap_malloc	heap_realloc	heap_space_unused
heap_switch	labs	lavg
lclip	lcount_ones	ldiv
llabs	llavg	llclip
llcount_ones	lldiv	llmax
llmin	lmax	lmin

Table 2-33: Library Functions in the stdlib.h Header File (Continued)

malloc	max	min
qsort	rand	realloc
space_unused	srand	strtod
strtol	strtoll	strtold
strtoul	strtoull	system

The ***Library Functions in the string.h Header File*** table lists the library functions in the `string.h` header file. Refer to string.h for more information on this header file.

Table 2-34: Library Functions in the string.h Header File

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

The ***Library Functions in the sys/adi_core.h Header File*** table lists functions in the `sys/adi_core.h` header file. Refer to sys/adi_core.h for more information on this header file.

Table 2-35: Library Functions in the sys/adi_core.h Header File

adi_core_enable	adi_core_id

The ***Library Functions in the time.h Header File*** table lists the library functions in the `time.h` header file. Refer to time.h for more information on this header file.

Table 2-36: 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.

Error Conditions

None

Example

```
#include <stdlib.h>

extern int errors;

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

See Also

raise, signal

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

None

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 `absfx` 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

None

Example

```
#include <stdfix.h>

long fract f;
f = abslr(0.75lr);      /* f == 0.75lr */
f = absfx(0.75lr);      /* f == 0.75lr */
```

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

adi_core_enable

Enable another core

Synopsis

```
#include <sys/adi_core.h>

ADI_CORE_ENABLE_STATUS adi_core_enable(ADI_CORE_ID id);
```

Description

The `adi_core_enable` function is available on multi-core processors, to release the other available cores. Once released, the core starts executing instructions from its reset address.

The `adi_core_enable` function takes a parameter of the enumeration type `ADI_CORE_ID` as the core ID of the core to enable. Currently this function should only be called from the first SHARC core (i.e. SHARC0) on targets with two SHARC cores, therefore the only valid argument is `ADI_CORE_SHARC1`. On success, the function returns `ADI_CORE_ENABLE_SUCCESS`.

Error Conditions

On failure, the `adi_core_enable` function returns one of the error codes defined in the `ADI_CORE_ENABLE_STATUS` enumeration. These are:

- `ADI_CORE_ENABLE_INVALID_CORE`: The core ID parameter did not specify one of the cores which may be enabled on the part.
- `ADI_CORE_ENABLE_CURRENT_CORE`: The core ID parameter specified the same core which is being used to execute the function.

Example

```
#include <sys/adi_core.h>

int main(void)
{
    // SHARC Core 1 is not enabled
```

```
    adi_core_enable(ADI_CORE_SHARC1);
    // SHARC Core 1 is now running
}
```

See Also

adi_core_id

adi_core_id

Identify caller's core

Synopsis

```
#include <sys/adi_core.h>

ADI_CORE_ID adi_core_id(void);
```

Description

The `adi_core_id` function returns an `ADI_CORE_ID` enumeration value indicating which processor core is executing the call to the function. This function is most useful on multi-core processors, when the caller is a function shared between both cores, but which needs to perform different actions (or access different data) depending on the core executing it.

The function returns the `ADI_CORE_SHARC0` value when executed by SHARC core 0, and a value of `ADI_CORE_SHARC1` when executed on SHARC core 1.

Error Conditions

None

Example

```
#include <sys/adi_core.h>

const char *core_name(void)
{
   if (adi_core_id() == ADI_CORE_SHARC0)
      return "SHARC Core 0";
   else
      return "SHARC Core 1";
}
```

See Also

adi_core_enable

adi_dump_all_heaps

Dump the current state of current heaps to a file

Synopsis

```
#include <heap_debug.h>

void adi_dump_all_heaps(char *filename);
```

Description

The `adi_dump_all_heaps` function writes the current state of all of the heaps known to the heap debugging library to the file specified by `filename`. The information written to the file consists of the address, size and state of any blocks on that heap that have been tracked by the heap debugging library, and the total memory currently allocated from that heap.

If the specified file exists, then the file is appended to; otherwise, a new file is created.

NOTE: The `adi_dump_all_heaps` function relies on the heap usage being tracked by the heap debugging library, any heap activity which is carried out when heap usage is not being tracked (when heap debugging is paused or disabled) will not be included in the output.

The `adi_dump_all_heaps` should be called only when it is safe to carry out I/O operations. Calling `adi_dump_all_heaps` from within an interrupt or an unscheduled region will result in adi_fatal_error being called.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

The `adi_dump_heap` function calls `adi_fatal_error` if it is unable to open the requested file.

Example

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

void dump_heaps()
{
  adi_dump_all_heaps("./dumpfile.txt");
}
```

See Also

adi_dump_heap, adi_fatal_error

adi_dump_heap

Dump the current state of a heap to a file

Synopsis

```
#include <heap_debug.h>

bool adi_dump_heap(char *filename, int heapindex);
```

Description

The `adi_dump_heap` function writes the current state of the heap identified by `heapindex` to the file specified by `filename`. The information written to the file consists of the address, size and state of any blocks on that heap tracked by the heap debugging library, and the total memory currently allocated from that heap.

If the specified file exists, then the file is appended to, otherwise a new file is created.

> **NOTE:** The `adi_dump_heap` function relies on the heap usage being tracked by the heap debugging library. Any heap activity which is carried out when heap usage is not being tracked (when heap debugging is paused or disabled) will not be included in the output.
>
> The `adi_dump_heap` function should be called only when it is safe to carry out I/O operations. Calling `adi_adi_dump_heap` from within an interrupt or an unscheduled region will result in adi_fatal_error being called.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

The `adi_dump_heap` function calls `adi_fatal_error` if it is unable to open the requested file.

Example

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

void dump_heap(int heapindex)
{
  if (!adi_dump_heap("./dumpfile.txt", heapindex)) {
    printf("heap %d does not exist\n", heapindex);
  }
}
```

See Also

adi_dump_all_heaps, adi_fatal_error

adi_fatal_error

Handle a non-recoverable error

Synopsis

```
#include <stdlib.h>

void adi_fatal_error(int general_code,
                     int specific_code,
                     int value);
```

Description

The `adi_fatal_error` function handles a non-recoverable error. The parameters `general_code`, `specific_code` and `value` will be written to global variables along with the return address, before looping around the label `__fatal_error`.

The `adi_fatal_error` function can be jumped to rather than called in order to preserve the return address if required.

See Fatal Error Handling for more information.

Error Conditions

None

Example

```
#include <stdlib.h>
#define MY_GENERAL_CODE (0x9)

void non_recoverable_error(int code, int value) {
  adi_fatal_error(MY_GENERAL_CODE, code, value);
}
```

See Also

adi_fatal_exception

adi_fatal_exception

Handle a non-recoverable exception

Synopsis

```
#include <stdlib.h>

void adi_fatal_exception(int general_code,
```

```
                              int specific_code,
                              int value);
```

Description

The `adi_fatal_exception` function handles a non-recoverable exception. The parameters `general_code`, `specific_code`, and `value` will be written to global variables along with the return address, before looping around the label `__fatal_exception`.

The `adi_fatal_exception` function can be jumped to rather than called in order to preserve the return address if required.

See Fatal Error Handling for more information.

Error Conditions

None

Example

```
#include <stdlib.h>

#define MY_GENERAL_CODE (0x9)

void non_recoverable_exception(int code, int value) {
   adi_fatal_exception(MY_GENERAL_CODE, code, value);
}
```

See Also

adi_fatal_error

adi_heap_debug_disable

Disable features of the heap debugging

Synopsis

```
#include <heap_debug.h>

void adi_heap_debug_disable(unsigned char flag);
```

Description

The `adi_heap_debug_disable` function accepts a bit-field parameter detailing which features are to be enabled. These bits are represented by macros defined in heap_debug.h.

These parameter bits can be combined using the bitwise OR operator to allow multiple settings to be disabled at once.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

None

Example

```
#include <heap_debug.h>

void disable_diagnostics()
{
   // Disable run-time errors
   adi_heap_debug_disable(_HEAP_STDERR_DIAG);
}
```

See Also

adi_heap_debug_enable

adi_heap_debug_enable

Enable features of the heap debugging

Synopsis

```
#include <heap_debug.h>

void adi_heap_debug_enable(unsigned char flag);
```

Description

The `adi_heap_debug_enable` function accepts a bit-field parameter detailing which features are to be enabled. These bits are represented by macros defined in heap_debug.h. `_HEAP_TRACK_USAGE` (track heap activity) is implicitly enabled when either `_HEAP_STDERR_DIAG` (generate diagnostics at runtime) or `_HEAP_HPL_GEN` (generate `.hpl` file of heap activity used by report) are enabled.

These parameter bits can be combined using the bitwise OR operator to allow multiple settings to be enabled at once.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

None

Example

```
#include <heap_debug.h>

void enable_hpl_gen()
{
   // Enable run-time errors and the generation of the .hpl file
   adi_heap_debug_enable(_HEAP_STDERR_DIAG | _HEAP_HPL_GEN);
}
```

See Also

adi_heap_debug_disable

adi_heap_debug_end

Finish heap debugging

Synopsis

```
#include <heap_debug.h>

void adi_heap_debug_end(void);
```

Description

The `adi_heap_debug_end` function records the end of the heap debugging.

Internal data used by the heap debugging library will be freed, the `.hpl` file generated will be closed (if `.hpl` generation is enabled) and any heap corruption or memory leaks will be reported. The `adi_heap_debug_end` function can be called multiple times, allowing heap debugging to be started and ended over specific sections of code.

Use `adi_heap_debug_end` in non-terminating applications to instruct the heap debugging library to carry out the end checks for the heap debugging in that application.

Do not call `adi_heap_debug_end` from within an ISR or when thread switching as there will be no way for it to produce any output.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

Corrupt blocks or memory leaks may be reported via the console view (if run-time diagnostics are enabled) or via the report (if `.hpl` file generation is enabled).

Example

```
#include <heap_debug.h>

void main_func()
{
   // Start heap debugging
   adi_heap_debug_enable(_HEAP_STDERR_DIAG);

   // Application code
   run_application();

   // Check for leaks or corruption
   adi_heap_debug_end();
}
```

See Also

adi_heap_debug_enable

adi_heap_debug_flush

Flush the heap debugging output buffer

Synopsis

```
#include <heap_debug.h>

void adi_heap_debug_flush(void);
```

Description

The `adi_heap_debug_flush` function flushes any buffered data to the `.hpl` file used by the reporter tool to generated the heap debugging report.

> **NOTE:** The `adi_heap_debug_flush` function should only be called when it is safe to carry out I/O operations. Calling `adi_heap_debug_flush` from within an interrupt or an unscheduled region will result in adi_fatal_error being called.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

The `adi_heap_debug_flush` function calls `adi_fatal_error` if called when it is unsafe to use I/O.

Example

```
#include <heap_debug.h>

void flush_hpl_buffer()
{
   adi_heap_debug_flush();
}
```

See Also

adi_fatal_error

adi_heap_debug_pause

Temporarily disable the heap debugging

Synopsis

```
#include <heap_debug.h>

void adi_heap_debug_pause(void);
```

Description

The `adi_heap_debug_pause` function disables the heap debugging functionality. When disabled, the heap debugging library has a minimal performance overhead compared to the non-debug versions of the heap debugging functions provided by the C/C++ run-time libraries. Pausing heap debugging means that any heap operations, which happen between pausing and re-enabling the heap debugging, will not be tracked, meaning that erroneous behavior may not be detected and false errors regarding unfreed blocks or unknown addresses may be reported.

Take care when using `adi_heap_debug_pause` in a threaded environment, as the heap debugging will be disabled globally rather than within the context of the current thread.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

None

Example

```
#include <heap_debug.h>
```

```
void a_performance_critical_function(void);

void performance_critical_fn_wrapper()
{
  adi_heap_debug_pause();
  a_performance_critical_function();
  adi_heap_debug_resume();
}
```

See Also

adi_heap_debug_resume

adi_heap_debug_reset_guard_region

Reset guard regions to default values

Synopsis

```
#include <heap_debug.h>

bool adi_heap_debug_reset_guard_region(void);
```

Description

The `adi_heap_debug_reset_guard_region` function resets the guard region values to the default. The heaps are checked for guard region corruption before all existing guard regions are replaced with the new values. If corruption is detected, then no guard regions are changed and `adi_heap_debug_reset_guard_region` returns false. The contents of existing allocated blocks are not changed, but any newly allocated blocks are pre-filled with the new allocated block pattern.

The default reset values are detailed in the ***Reset Values for Heap Guard Regions*** table.

Table 2-37: Reset Values for Heap Guard Regions

Region	*Value*
Free block	`0xBDBDBDBD`
Allocated block	`0xDDDDDDDD`
Block content (not `calloc`)	`0xEDEDEDED`

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

The `adi_heap_debug_reset_guard_region` function returns `false` if no guard region change was made, due to the detection of corruption on one of the heaps.

Example

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

void reset_guard_region()
{
   if (!adi_heap_debug_reset_guard_region()) {
      printf("couldn't reset guard regions\n");
   }
}
```

See Also

adi_heap_debug_set_guard_region

adi_heap_debug_resume

Re-enable the heap debugging

Synopsis

```
#include <heap_debug.h>

void adi_heap_debug_resume(void);
```

Description

The `adi_heap_debug_resume` function enables the heap debugging. Any allocations or de-allocations that occurred when the heap debugging was disabled will not have been tracked by the heap debugging library, so false errors regarding invalid addresses or memory leaks may be produced.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

None

Example

```
#include <heap_debug.h>

void a_performance_critical_function(void);
```

```
void performance_critical_fn_wrapper()
{
   adi_heap_debug_pause();
   a_performance_critical_function();
   adi_heap_debug_resume();
}
```

See Also

adi_heap_debug_pause

adi_heap_debug_set_buffer

Configure a buffer to be used by the heap debugging

Synopsis

```
#include <heap_debug.h>

bool adi_heap_debug_set_buffer(void *ptr, size_t size, size_t threshold);
```

Description

The `adi_heap_debug_set_buffer` function instructs the heap debugging library to use the specified buffer for the writing of the `.hpl` file used by the Reporter Tool to generate a heap debugging report. The buffer is of `size` addressable units starting at address `ptr`, with a flush threshold of `threshold` addressable units. The minimum size of the buffer in addressable units can be determined using the macro `_ADI_HEAP_MIN_BUFFER` (defined in heap_debug.h) and represents the memory required to store two entries of the heap debugging buffer along with associated call stacks. Changing the call stack depth after setting a buffer may alter the number of entries which can be held within the buffer.

Buffering can be disabled by calling `adi_heap_debug_set_buffer` with a null pointer as the first parameter.

Using a buffer will reduce the number of I/O operations to write the `.hpl` file to the host which should in turn result in a significant reduction in execution time when running applications which make frequent use of the heap.

If the buffer is full or no buffer is specified, and heap activity occurs where I/O is not permitted, that data will be lost.

The buffer will be flushed automatically when it is filled beyond a capacity threshold, specified by the `threshold` parameter, and it is safe to flush. Flushing can be triggered manually by calling adi_heap_debug_flush.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

NOTE: Only call `adi_heap_debug_set_buffer` when it is safe to carry out I/O operations. Calling `adi_heap_debug_set_buffer` from within an interrupt or an unscheduled region will result in adi_fatal_error being called.

Error Conditions

The `adi_heap_debug_set_buffer` function returns `false` if the buffer passed is not valid or big enough to be used the heap debugging library.

Example

```
#include <heap_debug.h>

char heapbuffer[1024];

bool set_buffer(void){
   if (sizeof(heapbuffer) < _ADI_HEAP_MIN_BUFFER) {
      return false;
}
return adi_heap_debug_set_buffer(&heapbuffer,
                                 sizeof(heapbuffer),
                                 sizeof(heapbuffer)/2);
}
```

See Also

No related functions

adi_heap_debug_set_call_stack_depth

Change the depth of the call stack recorded by the heap debugging library

Synopsis

```
#include <heap_debug.h>

bool adi_heap_debug_set_call_stack_depth(unsigned int depth);
```

Description

The `adi_heap_debug_set_call_stack_depth` function sets the maximum depth of the call stack recorded by the heap debugging library for use in the heap reports and diagnostic messages. The memory for the call stack is allocated from the system heap and requires memory of size `(2*sizeof(int))` per call stack element. The default value is 5 stack elements deep.

The `adi_heap_debug_set_call_stack_depth` function returns `true` if it is able to change the depth; otherwise, `false` is returned and the depth remains unchanged.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

The `adi_heap_debug_set_call_stack_depth` function returns `false` if it is unable to allocate sufficient memory for the new call stack.

Example

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

bool set_call_stack_depth(unsigned int size)
{
   if (!adi_heap_debug_set_call_stack_depth(size))
   {
       printf("unable to set heap debug call stack "
              "to %d elements\n", size);
       return false;
   } return true;
}
```

See Also

No related functions

adi_heap_debug_set_error

Change error types to be regarded as terminating errors

Synopsis

```
#include <heap_debug.h>

void adi_heap_debug_set_error(unsigned long flag);
```

Description

The `adi_heap_debug_set_error` function changes the severity of the specified types of heap error to a terminating run-time error. These types are represented as a bit-field using macros defined in heap_debug.h.

Terminating run-time errors print a diagnostic message to `stderr` before calling adi_fatal_error.

NOTE: Run-time errors need to be enabled for these changes to have any effect.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

None

Example

```
#include <heap_debug.h>

void set_errors()
{
   /* Enable run-time diagnostics */
   adi_heap_debug_enable(_HEAP_STDERR_DIAG);

   /* Regard frees from the wrong heap or of null pointers */
   /* as terminating run-time errors */
   adi_heap_debug_set_error(_HEAP_ERROR_WRONG_HEAP |
_HEAP_ERROR_NULL_PTR );}
```

See Also

adi_heap_debug_enable, adi_heap_debug_set_ignore, adi_heap_debug_set_warning

adi_heap_debug_set_guard_region

Changes the bit patterns written to guard regions around memory blocks

Synopsis

```
#include <heap_debug.h>

bool adi_heap_debug_set_guard_region(unsigned char free,
                                     unsigned char allocated,
                                     unsigned char content);
```

Description

The `adi_heap_debug_set_guard_region` function changes the bit pattern written to the guard regions around memory blocks used by the heap debugging library to check, if overwriting has occurred. The heaps are checked for guard region corruption before changing the guard regions. If any guard region is corrupt then `adi_heap_debug_set_guard_region` fails and the guard regions will not be changed. The contents of existing allocations are not be changed, but any new allocations will be pre-filled with the pattern specified by the `allocated` parameter.

The value of `free` is written to any free blocks, as well as the following guard region. Corruption of these blocks indicates that a pointer has been used to write to a block after it has been freed.

The value of `allocated` is written to the guard regions on either side of the allocated block. Corruption of these blocks indicates that overflow or underflow of that allocation has occurred.

The value of `content` is written to the allocated memory block, with the exception of memory allocated by `calloc`, which is zero filled. Seeing this value in live data indicates that memory allocated from the heap is used before being initialized.

The current values for the guard regions for free blocks, allocated blocks, and the pattern used for allocated block contents are stored in the `"C" char` variables `adi_heap_guard_free`, `adi_heap_guard_alloc`, and `adi_heap_guard_content`. These variables can be defined at build-time but should not be written to directly at run-time or false corruption errors may be reported.

The guard region values can be reset to the ADI default values by calling `adi_heap_debug_set_guard_region`.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

The `adi_heap_debug_set_guard_region` function returns false if it was unable to change the guard regions, due the presence of block corruption on one of the heaps.

Example

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

bool set_guard_regions()
{
   if (!adi_heap_debug_set_guard_region(0x11111111, 0x22222222, 0x33333333)
   {
      printf("failed to change guard regions\n");
      return false;
    }
    return true;
}
```

See Also

adi_heap_debug_reset_guard_region

adi_heap_debug_set_ignore

Change error types to be ignored

Synopsis

```
#include <heap_debug.h>

void adi_heap_debug_set_ignore(unsigned long flag);
```

Description

The `adi_heap_debug_set_ignore` function configures an error class as ignored. These types are represented as a bit-field using macros defined in heap_debug.h

Ignored errors produce no run-time diagnostics, but will appear in the heap debugging report (if generated).

NOTE: Run-time errors need to be enabled for these changes to have any effect.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

None

Example

```
#include <heap_debug.h>

void ignore_unwanted_errors()
{
  // Enable run-time diagnostics
  adi_heap_debug_enable(_HEAP_STDERR_DIAG);

  // Don't produce run-time diagnostics about frees
  // from the wrong heap or heap operations used
  // from within an interrupt
  adi_heap_debug_set_ignore(_HEAP_ERROR_WRONG_HEAP |
                            _HEAP_ERROR_IN_ISR);
}
```

See Also

adi_heap_debug_enable, adi_heap_debug_set_error, adi_heap_debug_set_warning

adi_heap_debug_set_warning

Change error types to be regarded as run-time warning

Synopsis

```
#include <heap_debug.h>

void adi_heap_debug_set_warning(unsigned long flag);
```

Description

The `adi_heap_debug_set_warning` function configures an error class to be regarded as a warning. These types are represented as a bit-field using macros defined in heap_debug.h.

A warning diagnostic is produced at runtime if an error of that class is detected, but the application will not terminate.

Any detected errors are recorded in the heap debugging report (if generated) as normal.

If the heap debugging library is unable to write a warning to `stderr` due to being in an interrupt or an unscheduled region, then the warning will be treated as an error and `adi_fatal_error` will be called. For this reason, setting `_HEAP_ERROR_IN_ISR` (heap usage within interrupt) to be a warning has no effect.

NOTE: Run-time errors need to be enabled for these changes to have any effect.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

None

Example

```
#include <heap_debug.h>

void set_warnings()
{
  // Enable run-time diagnostics
  adi_heap_debug_enable(_HEAP_STDERR_DIAG);

  // Produce warnings about de-allocating and
  // reallocating pointers not returned by an
  // allocation function and about de-allocations
  // not using functions which correspond to an
  // allocation, but don't terminate the application
  // on detection
  adi_heap_debug_set_warning(_HEAP_ERROR_INVALID_ADDRESS |
                             _HEAP_ERROR_FUNCTION_MISMATCH);
}
```

See Also

adi_heap_debug_enable, adi_heap_debug_set_error, adi_heap_debug_set_ignore

adi_libldr_Load

Load a loader stream for a dynamically-loadable module (DLM) from the given input stream.

Synopsis

```
#include <adi_libldr.h>

ADI_LIBLDR_RESULT adi_libldr_Load(FILE *fp);
```

Description

The `adi_libldr_Load` function is used to load a dynamically-loadable module (DLM) in loader stream form. The `fp` parameter gives the input stream from which the DLM loader stream should be read.

For more information on how to load a DLM loader stream using the `adi_libldr.h` header, see ***Dynamically-Loadable Modules*** in the ***System Runtime Documentation*** section of help.

Error Conditions

The `adi_libldr_Load` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
`ADI_LIBLDR_SUCCESS`	Success. The DLM was loaded correctly.
`ADI_LIBLDR_READ_ERROR`	`fread` failed to read expected amount of data from the input stream.
`ADI_LIBLDR_CHECKSUM_ERROR`	The checksum in the header block does not agree with the computed checksum.
`ADI_LIBLDR_SEEK_ERROR`	`fseek` failed to skip expected amount of data from the input stream.
`ADI_LIBLDR_UNSUPP_BLOCK`	The DLM loader stream contained an unsupported block type.

Example

```
#include <adi_libldr.h>

bool load_dlm(void)
{
   FILE *dlm = fopen("my_dlm.ldr", "rb");
   return (adi_libldr_Load(dlm) == ADI_LIBLDR_SUCCESS);
}
```

See Also

adi_libldr_LoadDbg, adi_libldr_Move, adi_libldr_MoveDbg, adi_libldr_LookUpSym

adi_libldr_LoadDbg

Read a loader stream for a dynamically-loadable module (DLM) from the given input stream, emitting any diagnostics to a log.

Synopsis

```
#include <adi_libldr.h>

ADI_LIBLDR_RESULT adi_libldr_LoadDbg(FILE *fp, FILE *log, bool do_copy);
```

Description

The `adi_libldr_LoadDbg` function is used to read a dynamically-loadable module (DLM) in loader stream form. The `fp` parameter gives the input stream from which the DLM loader stream should be read. The function also takes a `log` parameter which, if not `NULL`, provides an output stream to which any diagnostics are written. The `do_copy` parameter indicates whether to load the DLM to memory, or whether to perform a dry run, checking the DLM loader stream is valid without performing any other actions.

This function is provided to aid debugging; the equivalent `adi_libldr_Load` function should be used in production code.

For more information on how to load a DLM loader stream using the `adi_libldr.h` header, see ***Dynamically-Loadable Modules*** in the ***System Runtime Documentation*** section of help.

Error Conditions

The `adi_libldr_LoadDbg` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
`ADI_LIBLDR_SUCCESS`	Success. The DLM is loaded correctly.
`ADI_LIBLDR_READ_ERROR`	`fread` fails to read expected amount of data from the input stream.
`ADI_LIBLDR_CHECKSUM_ERROR`	The checksum in the header block does not agree with the computed checksum.
`ADI_LIBLDR_SEEK_ERROR`	`fseek` fails to skip expected amount of data from the input stream.
`ADI_LIBLDR_UNSUPP_BLOCK`	The DLM loader stream contained an unsupported block type.

Example

```
#include <adi_libldr.h>

/* read a loader stream from a file and check it is ok */
bool load_dlm_dry_run(void) {
   FILE *dlm = fopen("my_dlm.ldr", "rb");
   return (adi_libldr_LoadDbg(dlm, stdout, false) == ADI_LIBLDR_SUCCESS);
}
```

See Also

adi_libldr_Load, adi_libldr_Move, adi_libldr_MoveDbg, adi_libldr_LookUpSym

adi_libldr_LookUpSym

Look up a symbol in a given symbol table for a dynamically-loadable module (DLM) using a string key.

Synopsis

```
#include <adi_libldr.h>

ADI_LIBLDR_RESULT adi_libldr_LookUpSym(const char *symbol_key,
                                       const void *symbol_table,
                                       void      **symbol_address);
```

Description

The `adi_libldr_LookUpSym` function is used to query a symbol table associated with a loaded dynamically-loadable module (DLM). The `symbol_key` parameter is a unique string key that used in the symbol table to identify symbols in the DLM. The symbol_table is the address of the `symbol_table` provided by the loaded DLM. The address of the symbol, if found, is returned by storing it into the address given by the `symbol_address` parameter.

For more information on how to load a DLM loader stream using the `adi_libldr.h` header, see ***Dynamically-Loadable Modules*** in the ***System Runtime Documentation*** section of help.

Error Conditions

The `adi_libldr_LookUpSym` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
`ADI_LIBLDR_SUCCESS`	Success. The key was found in the provided symbol table and the corresponding address is stored in the address pointed to by the `symbol_address` parameter.
`ADI_LIBLDR_NULL_PTR`	The `symbol_key`, `symbol_table`, or `symbol_address` parameter is NULL.
`ADI_LIBLDR_BAD_SYMBOL`	The `symbol_key` parameter points to a zero-length string.
`ADI_LIBLDR_NO_SYMTAB`	The symbol table provided does not begin with the expected `ADI_LIBLDR_MAGIC_SYMNAME` and `ADI_LIBLDR_MAGIC_SYMADDR` constant pair that indicates a valid symbol table.
`ADI_LIBLDR_NO_SYMBOL`	The symbol key was not found in the symbol table.

Example

```
#include <adi_libldr.h>

int get_dlm_version(void) {
   void *version;
   if (adi_libldr_LookUpSym("version", dlm_symbol_table, &version) ==
                                                         ADI_LIBLDR_SUCCESS)
     return *((int *)version);
   return -1;
}
```

See Also

adi_libldr_Load, adi_libldr_LoadDbg, adi_libldr_Move, adi_libldr_MoveDbg

adi_libldr_Move

Load a loader stream for a dynamically-loadable module (DLM) in memory.

Synopsis

```
#include <adi_libldr.h>

ADI_LIBLDR_RESULT adi_libldr_Move(char_t *cp);
```

Description

The `adi_libldr_Move` function is used to load a dynamically-loadable module (DLM) in loader stream form. The `cp` parameter gives a pointer to a DLM loader stream in memory.

For more information on how to load a DLM loader stream using the `adi_libldr.h` header, see ***Dynamically-Loadable Modules*** in the ***System Runtime Documentation*** section of help.

Error Conditions

The `adi_libldr_Move` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
`ADI_LIBLDR_SUCCESS`	Success. The DLM was loaded correctly.
`ADI_LIBLDR_CHECKSUM_ERROR`	The checksum in the header block does not agree with the computed checksum.
`ADI_LIBLDR_UNSUPP_BLOCK`	The DLM loader stream contained an unsupported block type.

Example

```
#include <adi_libldr.h>

extern char_t *ldr_stream;
```

```
bool load_dlm(void) {
   return (adi_libldr_Move(ldr_stream) == ADI_LIBLDR_SUCCESS);
}
```

See Also

adi_libldr_Load, adi_libldr_LoadDbg, adi_libldr_MoveDbg, adi_libldr_LookUpSym

adi_libldr_MoveDbg

Read a loader stream for a dynamically-loadable module (DLM) in memory, emitting any diagnostics to a log.

Synopsis

```
#include <adi_libldr.h>

ADI_LIBLDR_RESULT adi_libldr_MoveDbg(char_t *cp, FILE *log, bool do_copy);
```

Description

The `adi_libldr_MoveDbg` function is used to read a dynamically-loadable module (DLM) in loader stream form. The `cp` parameter gives a pointer to a DLM loader stream in memory. The function also takes a `log` parameter which, if not `NULL`, provides an output stream to which any diagnostics are written. The `do_copy` parameter indicates whether to load the DLM to memory, or whether to perform a dry run, checking the DLM loader stream is valid without performing any other actions.

This function is provided to aid debugging; the equivalent `adi_libldr_Move` function should be used in production code.

For more information on how to load a DLM loader stream using the `adi_libldr.h` header, see ***Dynamically-Loadable Modules*** in the ***System Runtime Documentation*** section of help.

Error Conditions

The `adi_libldr_MoveDbg` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
ADI_LIBLDR_SUCCESS	Success. The DLM is loaded correctly.
ADI_LIBLDR_CHECKSUM_ERROR	The checksum in the header block does not agree with the computed checksum.
ADI_LIBLDR_UNSUPP_BLOCK	The DLM loader stream contained an unsupported block type.

Example

```
#include <adi_libldr.h>
extern char_t *ldr_stream;

/* read a loader stream from a file and check it is ok */
bool load_dlm_dry_run(void) {
   return (adi_libldr_MoveDbg(ldr_stream, stdout, false) == ADI_LIBLDR_SUCCESS);
}
```

See Also

adi_libldr_Load, adi_libldr_LoadDbg, adi_libldr_Move, adi_libldr_LookUpSym

adi_verify_all_heaps

Verify that no heaps contain corrupt blocks

Synopsis

```
#include <heap_debug.h>

bool adi_verify_all_heaps(void);
```

Description

The `adi_verify_all_heaps` function checks that each heap tracked by the heap debugging library contains no corrupted guard regions and that the underlying heap structure is correct. If a corrupt guard region is detected on any heaps then `adi_verify_all_heaps` will return `false`, otherwise `true` will be returned.

NOTE: The `adi_verify_all_heaps` function relies on the heap usage being tracked by the heap debugging library. Any heap activity carried out when heap usage is not being tracked (when heap debugging is paused or disabled) is not checked for corruption.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

The `adi_verify_all_heaps` function returns `false` if any corrupt guard regions are detected on any heap.

Example

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

void check_heaps()
{
    if (!adi_verify_all_heaps()) {
```

```
        printf("heaps contain corruption\n");
    }
    else {
        printf("heaps are ok\n");
    }
}
```

See Also

adi_verify_heap

adi_verify_heap

Verify that a heap contains no corrupt blocks

Synopsis

```
#include <heap_debug.h>

bool adi_verify_heap(int heapindex);
```

Description

The `adi_verify_heap` function checks that the heap specified with the index `heapindex` has no corrupt guard regions. If any guard region corruption is detected on that heap then `adi_verify_heap` returns `false`; otherwise, `true` is returned.

The heap index of static heaps can be identified by using heap_malloc. The heap index of a dynamically defined heap is the value returned from heap_install.

NOTE: The `adi_verify_heap` function relies on the heap usage being tracked by the heap debugging library. Any heap activity carried out when heap usage is not being tracked (when heap debugging is paused or disabled) is not be checked for corruption.

For more information on heap debugging, see ***Heap Debugging*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

The `adi_verify_heap` function returns `false` if any corrupt guard regions are detected on the specified heap.

Example

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

void check_heap(int heapindex)
```

```
{
  if (!adi_verify_heap(heapindex)) {
    printf("heap %d contain corruption\n", heapindex);
  } else {
       printf("heap %d is ok\n", heapindex);
  }
}
```

See Also

adi_verify_all_heaps

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, etc.)
- `MMM` is the month and will be of the form Jan, Feb, Mar, etc
- `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 hour, from 0 to 59
- `ss` is the second of the minute, 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

None

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 argument passed to them. The argument and returned result are in radians. The argument must be in the range [1, 1]. The output result will be in the range [-π/2, π/2].

Error Conditions

The arc sine functions return zero if the input is not in the range [-1, 1]. The arc sine functions do not update errno.

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 input argument. The argument and returned result are in radians. The output result is in the range [-π/2, π/2].

Error Conditions

None

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 `0` 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.

NOTE: 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 `0` 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) `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 `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 r1;

r1 = atold(" -123.4");       /* r1 = -123.4 */
r2 = atold("\t0x1.8p4");     /* r2 = 24.0   */
```

See Also

atoi, atol, 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.

NOTE: 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

None

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 `bitsfx` 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

None

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 `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_malloc, heap_realloc, malloc, realloc

carg

Real part of C99 complex value

Synopsis

```
#include <complex.h>

float cargf (complex float a);
double carg (complex double a);
long double cargl (complex long double a);
```

Description

The complex argument functions return the argument (or phase angle) of the complex input `a`.

These functions are only available when building in C99 mode.

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

None

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

cimag

Imaginary part of C99 complex value

Synopsis

```
#include <complex.h>

float cimagf (complex float a);
```

```
double cimag (complex double a);
long double cimagl (complex long double a);
```

Description

The complex imaginary functions return the imaginary part of the complex input `a`.

These functions are only available when building in C99 mode.

See Also

creal

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

None

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

None

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`. An alternative method of measuring the performance of an application is described in Measuring Cycle Counts.

Error Conditions

None

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 functions

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

None

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

llcount_ones, lcount_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 `countlsfx` 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

None

Example

```
#include <stdfix.h>

int n;
n = countlsulr(0.125ulr);        /* n == 2 */
n = countlsfx(0.125ulr);         /* n == 2 */
```

See Also

No related functions

creal

Real part of C99 complex value

Synopsis

```
#include <complex.h>

float crealf (complex float a);
```

```
double creal (complex double a);
long double creall (complex long double a);
```

Description

The complex real functions return the real part of the complex input `a`.

These functions are only available when building in C99 mode.

See Also

cimag

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

None

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

None

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 `divifx` family of functions computes the quotient and returns the closest integer value to the result.

Error Conditions

The `divifx` function has 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

dyn_AddHeap

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Specify a new region of target memory which may be used for relocated, dynamically-loaded code and data.

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_AddHeap(dyn_mem_image *image, dyn_heap *heap);
```

Description

The `dyn_AddHeap` function declares a new region of target memory that may be used to relocate the code or data in dynamically-loadable module (DLM) `image`, as previously validated by `dyn_ValidateImage`. The `heap` parameter indicates the width and alignment of the memory, as well as the start and size.

The heap parameter must point to a `dyn_heap` structure that has been initialized by `dyn_heap_init`.

Error Conditions

The `dyn_AddHeap` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
DYN_NO_ERROR	Success. The heap was added to the image's list of regions from which to allocate target memory.
DYN_BAD_PTR	Either image or heap was NULL.
DYN_BAD_WIDTH	A heap has already been specified which has the same width as the heap being added.

Example

```
#include <libdyn.h>

DYN_RESULT data_heap(dyn_mem_image *image) {,
               static int myspace[50];
               static dyn_heap h[1];
               dyn_heap_init(h, myspace, sizeof(myspace), 4, 2);
   /* error-checking omitted */
   return dyn_AddHeap(image, h);
}
```

See Also

dyn_ValidateImage, dyn_heap_init, dyn_SetSectionAddr, dyn_SetSectionMem, dyn_AllocSectionMemHeap, malloc

dyn_alloc

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Allocate space from a target heap

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_alloc(dyn_heap *heap,
                     size_t naddrs,
                     void **ptr);
```

Description

The `dyn_alloc` function allocates a number of contiguous addressable locations from the target heap specified by the `heap` parameter. The first of these allocated locations is returned as the address pointed-to by the `ptr` parameter. The `naddrs` parameter indicates how many contiguous locations must be allocated.

This function is not normally called directly; it is used by `dyn_AllocSectionMem` and `dyn_AllocSectionMemHeap`.

Error Conditions

The `dyn_alloc` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	Reason
`DYN_NO_ERROR`	Success. The space was allocated.
`DYN_BAD_PTR`	Either ptr or heap was `NULL`.
`DYN_BAD_IMAGE`	The available space in the heap is not aligned according to the heap's alignment. This should never occur.
`DYN_TOO_SMALL`	There is insufficient space left in the heap to allocate naddrs locations in an aligned manner.

Example

```
#include <libdyn.h>

void *get_space(dyn_heap *heap)
{
   void *ptr = 0;
   if (dyn_alloc(heap, 100, &ptr) == DYN_NO_ERROR)
      return 0;
      return ptr;
}
```

See Also

dyn_ValidateImage, dyn_heap_init, dyn_AddHeap, dyn_Relocate, dyn_FreeSectionMem, dyn_AllocSectionMem-Heap, malloc

dyn_AllocSectionMem

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Allocate target memory aligned for a section in a dynamically-loadable module

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_AllocSectionMem(dyn_mem_image *image,
                               dyn_section *sections,
                               size_t secnum,
                               dyn_section_mem **mem);
```

Description

The dyn_AllocSectionMem function allocates a target memory buffer large enough to hold the contents of section secnum, in dynamically-loadable module (DLM) image, as previously validated by dyn_ValidateImage. The sections parameter is a local copy of the DLM's section table, obtained by dyn_GetSections. The memory allocated by this function should be freed in a single step at a later time, by calling dyn_FreeSectionMem.

Two areas of memory are allocated by this function:

1. A space is allocated in target memory to hold the contents of the section. This space is allocated by dyn_alloc from a heap defined by dyn_AddHeap; the heap in question is selected on the basis of the memory width of the section secnum, by the dyn_GetHeapForWidth function.
2. A space is allocated in local memory to keep track of this allocation. This memory is allocated from the default heap, and is attached to image, so that it may be freed later.

On exit, *mem points to the second of the two allocations.

Error Conditions

The dyn_AllocSectionMem function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
DYN_NO_ERROR	Success. *mem contains a pointer to a suitable block of memory; mem->aligned_addr can be used by dyn_SetSectionAddr for section secnum.
DYN_BAD_PTR	One or more of the pointer parameters was NULL.
DYN_NO_MEM	malloc failed, when attempting to allocate sufficient memory.
DYN_BAD_IMAGE	The secnum parameter does not refer to a valid section in the DLM.

Example

```
#include <libdyn.h>

dyn_section_mem *secmem(dyn_mem_image *image,
                        dyn_section *sections,
                        int nsecs) {
   int i;
```

```
   dyn_section_mem *mem = 0;
   for (i = 0; i < nsecs; i++) {
      if (dyn_AllocSectionMem(image, sections, i, &mem) != DYN_NO_ERROR)
         return NULL;
   }
   return mem;
}
```

See Also

dyn_AddHeap, dyn_ValidateImage, dyn_alloc, dyn_GetHeapForWidth, dyn_Relocate, dyn_FreeSectionMem, dyn_AllocSectionMemHeap, malloc

dyn_AllocSectionMemHeap

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Allocate memory from a given heap, aligned for a section in a dynamically-loadable module

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_AllocSectionMemHeap(dyn_mem_image *image,
                                   dyn_section *sections,
                                   size_t secnum,
                                   dyn_section_mem **mem,
                                   int heapidx);
```

Description

The `dyn_AllocSectionMemHeap` function allocates a target memory buffer large enough to hold the contents of section `secnum`, in dynamically-loadable module (DLM) `image`, as previously validated by `dyn_ValidateImage` . The sections parameter is a local copy of the DLM's section table, obtained by `dyn_GetSections`. The memory allocated by this function should be freed in a single step at a later time, by calling `dyn_FreeSectionMem`. The `heapidx` parameter indicates which heap should be used to allocate housekeeping space.

Two areas of memory are allocated by this function:

1. A space is allocated in target memory to hold the contents of the section. This space is allocated by `dyn_alloc` from a heap defined by `dyn_AddHeap`; the heap in question is selected on the basis of the memory width of the section `secnum` by `dyn_GetHeapForWidth`.

2. A space is allocated in local memory to keep track of this allocation. This memory is allocated using `heap_malloc`, with the heap in question specified by `heapidx`. The resulting memory is attached to `image`, so that it may be freed later.

On exit, `*mem` points to the second of the two allocations.

Error Conditions

The `dyn_AllocSectionMemHeap` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
DYN_NO_ERROR	Success. `*mem` contains a pointer to a suitable block of memory; `mem->aligned_addr` can be used by `dyn_SetSectionAddr` for section `secnum`.
DYN_BAD_PTR	One or more of the pointer parameters was `NULL`.
DYN_NO_MEM	`malloc` failed, when attempting to allocate sufficient memory.
DYN_BAD_IMAGE	The `secnum` parameter does not refer to a valid section in the DLM.

Example

```
#include <libdyn.h>

dyn_section_mem *secmem(dyn_mem_image *image,
                        dyn_section *sections,
                        int nsecs) {
   int i;
   dyn_section_mem *mem = 0;
   for (i = 0; i < nsecs; i++) {
      if (dyn_AllocSectionMemHeap(image, sections, i, &mem, 0) != DYN_NO_ERROR)
         return NULL;
   }
   return mem;
}
```

See Also

dyn_AddHeap, dyn_ValidateImage, dyn_alloc, dyn_GetHeapForWidth, dyn_Relocate, dyn_FreeSectionMem, dyn_AllocSectionMemHeap, malloc

dyn_CopySectionContents

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Copy the sections of a valid dynamically-loadable module

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_CopySectionContents(dyn_mem_image *image, dyn_section *sections);
```

Description

The `dyn_CopySectionContents` function will copy the contents of all sections from a dynamically-loadable module (DLM), into previously-allocated local space. `image` is a DLM previously validated by `dyn_ValidateImage`, and `sections` is a local copy of the DLM's section table, obtained by `dyn_GetSections`. An address must have previously been allocated to each section, by `dyn_SetSectionAddr`.

Error Conditions

The `dyn_CopySectionContents` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
`DYN_NO_ERROR`	Success. The DLM section contents were copied.
`DYN_BAD_PTR`	The sections or image parameter is `NULL`.
`DYN_BAD_IMAGE`	The image does not have the right magic number, or offsets within the image are nonsensical.

Example

```
#include <libdyn.h>

int copy_dlm(dyn_mem_image *image, dyn_sections *secs) {
   if (dyn_CopySectionContents(image, secs) == DYN_NO_ERROR)
      return 0;
   return -1;
}
```

See Also

dyn_ValidateImage, dyn_GetNumSections, dyn_GetSections, dyn_GetStringTableSize, dyn_GetStringTable, dyn_GetExpSymTab, dyn_LookupByName, dyn_Relocate, dyn_SetSectionAddr, dyn_AllocSectionMem

dyn_FreeEntryPointArray

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. See ***Dynamically-Loadable Modules*** in the ***System Runtime***

Documentation section of help for information on how to use the APIs in adi_libldr.h to load a DLM from a loader stream.

Release a previously-allocated list of entry points to the dynamically-loadable module

Synopsis

```
#include <libdyn.h>

void dyn_FreeEntryPointArray(char *strtab, char **entries);
```

Description

The `dyn_FreeEntryPointArray` function releases memory that is allocated by `dyn_GetEntryPointArray`.

Error Conditions

None

Example

See dyn_GetEntryPointArray for an example.

See Also

dyn_ValidateImage, dyn_GetExpSymTab, dyn_LookupByName, dyn_Relocate, dyn_GetEntryPointArray

dyn_FreeSectionMem

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Release memory allocated for sections in a dynamically-loadable module

Synopsis

```
#include <libdyn.h>

void dyn_FreeSectionMem(dyn_mem_image *image);
```

Description

The `dyn_FreeSectionMem` function releases house-keeping memory blocks that were allocated by `dyn_AllocSectionMem` or `dyn_AllocSectionMemHeap`. `image` is a DLM previously validated by `dyn_ValidateImage`. Target memory, allocated from heaps declared by `dyn_AddHeap`, remains valid.

Error Conditions

None

Example

```
#include <libdyn.h>

void secmem(dyn_mem_image *image, dyn_section *sections ,int nsecs) {
   int i;
   dyn_section_mem *mem = 0;
   for (i = 0; i < nsecs; i++) {
      if (dyn_AllocSectionMem(image, sections, i, &mem) != DYN_NO_ERROR)
          return;
   }
   do_something();
   dyn_FreeSectionMem(image);
   return;
}
```

See Also

dyn_ValidateImage, dyn_GetNumSections, dyn_GetSections, dyn_GetStringTableSize, dyn_GetStringTable, dyn_GetExpSymTab, dyn_LookupByName, dyn_Relocate, dyn_SetSectionAddr, dyn_CopySectionContents, dyn_AllocSectionMem

dyn_GetEntryPointArray

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Obtain a list of symbols exported by a dynamically-loadable module

Synopsis

```
#include <libdyn.h>

_RESULT dyn_GetEntryPointArray(dyn_mem_image *image,
                               size_t symidx,
                               size_t stridx,
```

```
                              char **hstrtab,
                              char ***entries,
                              size_t *num_entries);
```

Description

The `dyn_GetEntryPointArray` function obtains the contents of the exported symbol table of the dynamically-loadable module (DLM) `image`, in an array of string pointers, pointed to by `*entries`. `*num_entries` is set to contain the number of entries in the allocated array. Each entry in the allocated array points to a string in a local copy of the string table, converted to local string format. `*entries` is set to point to this local string table.

This function can be used to determine which symbols are exported by the DLM, if this is not known in advance. Once the array of entry-point strings has been obtained, the strings can be passed to `dyn_LookupByName` to determine the resolved address of the entry-point.

This function may only be called after the DLM has been relocated by calling `dyn_Relocate`; prior to that point, the exported symbol table's entries are not completely resolved.

The `symidx` and `stridx` parameters identify the sections that contain the exported symbol table and exported string table, respectively; these parameters are obtained via `dyn_GetExpSymTab`.

The allocated memory should be freed by `dyn_FreeEntryPointArray`, once it is no longer required.

Error Conditions

The `dyn_GetEntryPointArray` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
`DYN_NO_ERROR`	Success. `*ptr` contains the address of the symbol, in the relocated image.
`DYN_BAD_PTR`	One or more of the pointer parameters is `NULL`.
`DYN_NO_MEM`	There was not enough space to allocate either the entry array, or the local copy of the string table.
`DYN_NOT_FOUND`	The sections for the exported string table or exported symbol table could not be retrieved.

Example

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

void list_syms(dyn_mem_image *image,
               const char *strtab,
               dyn_section *sections) {
   size_t symidx, stridx;
   char *hstrtab, **syms;
   int i, nsyms;
   dyn_GetExpSymTab(image, symtab, sections, &symidx, &stridx);
                    dyn_GetEntryPointArray(image, symidx, stridx,
```

```
                  &hstrtab, &nsyms, &syms);
   for (i = 0; i < nsyms; i++)
      printf("Sym %d is %s\n", i, syms[i]);
   dyn_FreeEntryPointArray(hstrtab, syms);
}
```

See Also

dyn_ValidateImage, dyn_GetNumSections, dyn_GetSections, dyn_GetStringTableSize, dyn_GetStringTable, dyn_GetExpSymTab, dyn_Relocate, dyn_SetSectionAddr, dyn_CopySectionContents, dyn_AllocSectionMem

dyn_GetExpSymTab

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Locate a dynamically-loadable module's table of exported symbols

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_GetExpSymTab(dyn_mem_image *image,
                            const char *strtab,
                            dyn_section *sections,
                            size_t *symidx,
                            size_t *stridx);
```

Description

The `dyn_GetExpSymTab` function searches the dynamically-loadable module (DLM) pointed to by `image`, looking for the table of exported symbols. The `strtab` and `sections` parameters must be pointers to the DLM's string table and section table, obtained by `dyn_GetStringTable` and `dyn_GetSections`, respectively.

The DLM's exported-symbol table consists of two sections. One is a string table, containing the names of exported symbols in native processor format; the other is a table where each entry points to the symbol's name in said string table, and to the symbol itself (whether code or data).

If successful, the function records the section numbers of the exported section table and exported string table into the locations pointed to by `symidx` and `stridx`, respectively.

Error Conditions

The `dyn_GetExpSymTab` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
DYN_NO_ERROR	Success. *symidx contains the section number containing the exported symbol table, and *stridx contains the section number containing the exported string table.
DYN_BAD_PTR	One or more of the parameters is NULL.
DYN_BAD_IMAGE	The function could not locate sections for both the exported string table and the exported symbol table.

Example

```
#include <libdyn.h>

static size_t sec_tab, str_tab;

int find_secs(dyn_mem_image *image,
              const char strtab,
              dyn_section *sections) {
   if (dyn_GetExpSymTab(image, strtab, sections,
                        &sec_tab, &str_tab) == DYN_NO_ERROR)
      return 0;
   return -1;
}
```

See Also

dyn_ValidateImage, dyn_GetNumSections, dyn_GetSections, dyn_GetStringTableSize, dyn_GetStringTable, dyn_LookupByName, dyn_Relocate, dyn_SetSectionAddr, dyn_CopySectionContents, dyn_AllocSectionMem

dyn_GetHeapForWidth

ATTENTION: Support for dynamically-loadable modules (DLMs) using libdyn.h is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Locate a target-memory heap that has the right number of bits per addressable unit.

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_GetHeapForWidth(dyn_mem_image *image,
                               size_t byte_width,
                               dyn_heap **heap);
```

Description

The dyn_GetHeapForWidth function searches all target-memory heaps that have been declared for this image (using the dyn_AddHeap function), and returns the one that has a width of byte_width via *heap, if there is one.

Error Conditions

The dyn_GetHeapForWidth function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
DYN_NO_ERROR	Success. *heap contains a pointer to a heap which may be used for allocation.
DYN_BAD_PTR	Either heap or image was NULL.
DYN_NOT_FOUND	No heap has been attached to image using dyn_AddHeap(), which has a width that matches byte_width.

Example

```
#include <libdyn.h>

dyn_heap *fetch_heap(dyn_mem_image *image, size_t width) {
   dyn_heap *heap = 0;
   if (dyn_GetHeapForWidth(image, &heap) != DYN_NO_ERROR)
      return NULL;
   return heap;
}
```

See Also

dyn_AddHeap, dyn_ValidateImage, dyn_heap_init, dyn_alloc, dyn_FreeSectionMem, dyn_AllocSectionMemHeap, malloc

dyn_GetNumSections

ATTENTION: Support for dynamically-loadable modules (DLMs) using libdyn.h is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Obtain the number of sections in a dynamically-loadable module

Synopsis

```
#include <libdyn.h>
```

```
DYN_RESULT dyn_GetNumSections(dyn_mem_image *image,
                              size_t *num_sections);
```

Description

The `dyn_GetNumSections` function returns the number of sections in a validate dynamically-loadable module (DLM), as produced by `elf2dyn`. The `image` parameter should have been populated by a previous call to `dyn_ValidateImage`.

In the context of this function, "sections" means "portions of the DLM that contain executable code or usable data"; it does not include the string table or any relocation for the DLM.

Upon success, the function writes the number of sections to the location pointed to by the `num_sections` parameter.

Error Conditions

The `dyn_GetNumSections` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
`DYN_NO_ERROR`	Success. `*num_sections` will contain the section count.
`DYN_BAD_PTR`	The `image` or `num_sections` parameter is `NULL`.

Example

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

void count_sections(dyn_mem_image *dlm_info) {
   size_t nsec;
   if (dyn_GetNumSections(dlm_info, &nsec) == DYN_NO_ERROR)
      printf("There are %d section\n", nsec);
}
```

See Also

dyn_ValidateImage, dyn_GetSections, dyn_GetStringTableSize, dyn_GetStringTable, dyn_GetExpSymTab, dyn_LookupByName, dyn_Relocate, dyn_SetSectionAddr, dyn_AllocSectionMem, dyn_CopySectionContents

dyn_GetSections

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Obtain a native copy of the section table from a valid dynamically-loadable module.

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_GetSections(dyn_mem_image *image,
                           dyn_section *sections);
```

Description

The `dyn_GetSections` function accepts a pointer `sections` to a block of memory, and populates it with a native copy of the section table from the dynamically-loadable module (DLM) pointed to by `image`. The resulting section table copy is in the native byte order of the target processor.

The memory buffer must have been allocated previously, and must be large enough to contain all the section headers for the DLM.

Error Conditions

The `dyn_GetSections` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
`DYN_NO_ERROR`	Success. The section table will copied to sections.
`DYN_BAD_PTR`	The sections or image parameter is `NULL`.

Example

```
#include <stdlib.h>
#include <libdyn.h>

char *get_sec_table(dyn_mem_image *image, int nsecs) {
   char *space = malloc(nsecs * sizeof(dyn_section));
   if (dyn_GetSections(image, space) == DYN_NO_ERROR)
      return space;
   return NULL;
}
```

See Also

dyn_ValidateImage, dyn_GetNumSections, dyn_GetStringTableSize, dyn_GetStringTable, dyn_GetExpSymTab, dyn_LookupByName, dyn_Relocate, dyn_SetSectionAddr, dyn_AllocSectionMem, dyn_CopySectionContents

dyn_GetStringTable

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Obtain a native copy of the string table of a valid dynamically-loadable module

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_GetStringTable(dyn_mem_image *image,
                              char *buffer);
```

Description

The `dyn_GetStringTable` function copies the string table from the dynamically-loadable module image to the space pointed to by buffer. The resulting copy is in the native format of the target processor.

Error Conditions

The `dyn_GetStringTable` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
`DYN_NO_ERROR`	Success. buffer contains a native copy of the string table (one character per location).
`DYN_BAD_PTR`	The buffer or image parameter is `NULL`.

Example

```
#include <stdlib.h>
#include <libdyn.h>

char *get_strtab(dyn_mem_image *dlm_info, size_t *nchars) {
   char *ptr = malloc(nchars);
   if (dyn_GetStringTable(dlm_info, ptr) == DYN_NO_ERROR)
      return ptr;
   return NULL;
}
```

See Also

dyn_ValidateImage, dyn_GetNumSections, dyn_GetSections, dyn_GetStringTableSize, dyn_GetExpSymTab, dyn_LookupByName, dyn_Relocate, dyn_SetSectionAddr, dyn_AllocSectionMem, dyn_CopySectionContents

dyn_GetStringTableSize

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Get the size of the string table in a valid dynamically-loadable module

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_GetStringTableSize(dyn_mem_image *image,
                                  size_t *sz);
```

Description

The `dyn_GetStringTableSize` function returns the number of bytes required to hold the string table for the dynamically-loadable module (DLM) pointed to by `image`. The size is returned in the location pointed to by the `sz` parameter.

In a dynamically-loadable module, the string table contains the names of the various sections in the DLM. It does *not* contain character strings or other data that constitutes the loadable part of the DLM.

Error Conditions

The `dyn_GetStringTableSize` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
`DYN_NO_ERROR`	Success. `*sz` contains the size of the string table.
`DYN_BAD_PTR`	The `sz` or `image` parameter is `NULL`.

Example

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

void get_strtab_size(dyn_mem_image *dlm_info) {
   size_t nchars;
   if (dyn_GetStringTableSize(dlm_info, &nchars) == DYN_NO_ERROR)
      printf("There are %d characters in the table\n", nchars);
}
```

See Also

dyn_ValidateImage, dyn_GetNumSections, dyn_GetSections, dyn_GetStringTableSize, dyn_GetExpSymTab, dyn_LookupByName, dyn_Relocate, dyn_SetSectionAddr, dyn_AllocSectionMem, dyn_CopySectionContents

dyn_heap_init

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Initialize a target heap for dynamically-loadable modules

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_heap_init(dyn_heap *heap,
                         void *base,
                         size_t size,
                         size_t width,
                         size_t align);
```

Description

The `dyn_heap_init` function initializes the heap parameter, so that it contains a description of a region of target memory that can be used to relocate dynamically-loaded code or data. The resulting structure will be suitable for passing to `dyn_AddHeap`.

The heap parameter must point to a `dyn_heap` structure that is initialized as follows:

- `base` - the address of the first addressable unit in the region of target memory.
- `size` - the number of addressable units that can be allocated. Therefore, this should be set to the same value as `total_size`.
- `width` - should be set to the number of 8-bit values that can fit into a single location in the target memory. Therefore: 2 for VISA space, 4 for normal data memory, 6 for program memory, and 8 for long-word data memory. Note that only one heap may be specified, for each given width.
- `align` - when memory is allocated from this region, the offset into the region will be a multiple of this value. Therefore, this must be 1, 2 or 4, as required for memory alignment.

Error Conditions

The `dyn_heap_init` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
DYN_NO_ERROR	Success. The dyn_heap structure is now initialized.
DYN_BAD_PTR	Either image or heap was NULL, or size was zero.
DYN_BAD_IMAGE	The base pointer was not appropriately aligned for the align parameter.

Example

```
#include <libdyn.h>

DYN_RESULT data_heap(dyn_heap *heap) {,
    static int myspace[50];
    return dyn_heap_init(heap, myspace, sizeof(myspace), 4, 2);
}
```

See Also

dyn_ValidateImage, dyn_GetNumSections, dyn_GetSections, dyn_GetStringTableSize, dyn_GetExpSymTab, dyn_LookupByName, dyn_Relocate, dyn_SetSectionAddr, dyn_CopySectionContents, dyn_AllocSectionMem, malloc

dyn_LookupByName

ATTENTION: Support for dynamically-loadable modules (DLMs) using libdyn.h is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Locate an exported symbol in a dynamically-loadable module

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_LookupByName(dyn_mem_image *image,
                            const char *name,
                            void *symtab,
                            uint32_t secsize,
                            void **ptr);
```

Description

The dyn_LookupByName function searches the exported symbol table of the dynamically-loadable module (DLM) image, looking for a symbol called name. If such a symbol is found, the symbol's address is returned in the location pointed to by ptr. symtab is a pointer to the contents of the DLM's exported symbol table, as previously located via dyn_GetExpSymTab; secsize indicates the section's size.

This function may only be called after the DLM has been relocated by calling `dyn_Relocate`; prior to that point, the exported symbol table's entries are not completely resolved.

The `name` parameter must match the exported symbol exactly. This means that it must also be mangled appropriately for the symbol's namespace.

Error Conditions

The `dyn_LookupByName` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
DYN_NO_ERROR	Success. `*ptr` contains the address of the symbol, in the relocated image.
DYN_BAD_PTR	The `ptr` or `image` parameter is `NULL`.
DYN_NOT_FOUND	The exported symbol table does not contain a symbol whose name exactly matches `name`.

Example

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

int call_fn(dyn_mem_image *image,
            void *symtab,
            uint32_t secsize,
            const char *fnname) {
   void *ptr;
   if (dyn_LookupByName(image, name, symtab,
                        secsize, &ptr) == DYN_NO_ERROR) {
      int (*fnptr)(void) = (int (*)(void))*ptr;
      return (*fnptr)();
   }
   return -1;
}
```

See Also

dyn_ValidateImage, dyn_GetNumSections, dyn_GetSections, dyn_GetStringTableSize, dyn_GetExpSymTab, dyn_Relocate, dyn_SetSectionAddr, dyn_AllocSectionMem, dyn_CopySectionContents

dyn_RecordRelocOutOfRange

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Record which relocation cannot be completed, while relocating a dynamically-loadable module

Synopsis

```
#include <libdyn.h>

int dyn_RecordRelocOutOfRange(void *ref_addr,
                              uint32_t sym_addr);
```

Description

The `dyn_RecordRelocOutOfRange` function is invoked by `dyn_Relocate`, if a computed relocation is out of range. It provides an opportunity to make a note of the offending reference. Alternatively, it provides an opportunity to ignore the problem.

`ref_addr` is the target address of the location being relocated, while `sym_addr` is the computed location or value which is being referenced by `ref_addr`. `sym_addr` is presented before being manipulated to fit into the field at `ref_addr`. For example, if `ref_addr` only references even addresses, the stored value in the field might be shifted down one place; `sym_addr` represents the value before this shift has happened.

The default implementation of the `dyn_RecordRelocOutOfRange` function records both `ref_addr` and `sym_addr`, so that they can be retrieved later using `dyn_RecordRelocOutOfRange`.

Error Conditions

The `dyn_RecordRelocOutOfRange` function must return a value indicating whether this combination of `ref_addr` and `sym_addr` should be considered an error. If the function returns `false`, then `dyn_Relocate` will continue its operation. If the function returns `true`, then `dyn_Relocate` will abort.

Example

```
#include <libdyn.h>

int dyn_RecordRelocOutOfRange(void *ref_addr, uint32_t sym_addr) {
   /* alternative implementation that ignores all errors */
   return 0;
}
```

See Also

dyn_Relocate, dyn_RetrieveRelocOutOfRange

dyn_Relocate

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Relocate a dynamically-loadable module

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_Relocate(dyn_mem_image *image, dyn_section *sections);
```

Description

The `dyn_Relocate` function processes the relocations in a dynamically-loadable module (DLM) once its sections have been copied into local memory.

`image` is the DLM, as loaded and validated. `sections` is a copy of the DLM's section table, as obtained via `dyn_GetSections`. Before relocation can be performed, space must have been allocated for each of the sections in the file, using `dyn_AllocSectionMem`, and the sections' contents copied into that space using `dyn_CopySectionContents`.

Error Conditions

The `dyn_Relocate` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
`DYN_NO_ERROR`	Success. All sections were relocated.
`DYN_BAD_PTR`	The sections or image parameter is `NULL`.
`DYN_NO_SECTION_ADDR`	There is a section in the DLM which has not had an address allocated, prior to attempting to relocate it.
`DYN_BAD_RELOC`	The DLM contains a relocation that is not recognized by the current instance of libdyn.
`DYN_BAD_WIDTH`	The DLM contains a relocation that references a section with a word size not supported by this instance of libdyn.
`DYN_NOT_ALIGNED`	The DLM could not complete relocations because there is a section that is not appropriately aligned for its word size.
`DYN_OUT_OF_RANGE`	The DLM could not apply a relocation because the computed value does not fit into the available space. This generally means that the reference and the target of the relocation are too far apart. The function will invoke dyn_RetrieveRelocOutOfRange to record the details of the failing relocation. These details can be retrieved with dyn_RetrieveRelocOutOfRange.

Example

```
#include <libdyn.h>

int reloc_dlm(dyn_mem_image *dlm_info, dyn_section *sections) {
    if (dyn_Relocate(dlm_info, sections) == DYN_NO_ERROR)
       return 0;
    return -1;
}
```

See Also

dyn_ValidateImage, dyn_GetNumSections, dyn_GetSections, dyn_GetStringTableSize, dyn_GetExpSymTab, dyn_LookupByName, dyn_SetSectionAddr, dyn_AllocSectionMemHeap, dyn_CopySectionContents, dyn_RecordRelocOutOfRange, dyn_RetrieveRelocOutOfRange

dyn_RetrieveRelocOutOfRange

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Retrieve information about a relocation that failed

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_RetrieveRelocOutOfRange(void **ref_addr, uint32_t *sym_addr);
```

Description

The `dyn_RetrieveRelocOutOfRange` function is used to retrieve information about a failing relocation, if `dyn_Relocate` returns `DYN_OUT_OF_RANGE`. The information must first have been saved by `dyn_RecordRelocOutOfRange`.

`*ref_addr` will be set to the target address of the location that was being relocated, while `*sym_addr` will be set to the computed location or value that was being referenced by `*ref_addr`r.

Error Conditions

The `dyn_RetrieveRelocOutOfRange` function returns a value to indicate the status of its operation, as follows.

Returned Value	*Reason*
`DYN_NO_ERROR`	Success. `*ref_addr` and `*sym_addr` have been updated.
`DYN_BAD_PTR`	Either `ref_addr` or `sym_addr` was `NULL`.

Example

```
#include <libdyn.h>

void reloc_dlm(dyn_mem_image *dlm_info, dyn_section *sections) {
   if (dyn_Relocate(dlm_info, sections) == DYN_OUT_OF_RANGE &&
      dyn_RetrieveRelocOutOfRange(&ref, &sym) == DYN_NO_ERROR)
```

```
   printf("Relocation %p -> %p failed\n", ref, sym);
}
```

See Also

dyn_Relocate, dyn_RetrieveRelocOutOfRange

dyn_RewriteImageToFile

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Write a dynamically-loadable module back to a file, after relocation

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_RewriteImageToFile(dyn_mem_image *image,
                                  dyn_section *sections,
                                  size_t num_sections,
                                  FILE *outf);
```

Description

The `dyn_RewriteImageToFile` function writes the contents of a dynamically-loadable module (DLM) to the specified output stream `outf`, after relocation has taken place.

`image` is the DLM, as loaded, validated and relocated. `sections` is a copy of the DLM's section table, as obtained via `dyn_GetSections`.

Error Conditions

The `dyn_RewriteImageToFile` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
`DYN_NO_ERROR`	Success. All sections were written back to the output stream without error.
`DYN_BAD_WRITE`	One of the output operations on the output stream did not succeed.
`DYN_NO_MEM`	There was insufficient memory to obtain a local working copy of some data.
`DYN_BAD_PTR`	The image parameter was `NULL`, or a there is a corrupt internal memory reference.
`DYN_NOT_FOUND`	Not all sections could be located, suggesting that the `num_sections` parameter is incorrect.

Example

```
#include <libdyn.h>

int reloc_dlm(dyn_mem_image *dlm,
              dyn_section *secs,
              size_t nsecs,
              FILE *fp) {
   if (dyn_Relocate(dlm, secs) == DYN_NO_ERROR &&
       dyn_RewriteImageToFile(dlm, secs, nsecs, fp) == DYN_NO_ERROR)
       return 0;
   return -1;
}
```

See Also

dyn_ValidateImage, dyn_GetNumSections, dyn_GetSections, dyn_GetStringTableSize, dyn_GetStringTable, dyn_GetExpSymTab, dyn_LookupByName, dyn_SetSectionAddr, dyn_AllocSectionMem, dyn_CopySectionContents

dyn_SetSectionAddr

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Set the local address for a section in a dynamically-loadable module

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_SetSectionAddr(dyn_mem_image *image,
                              dyn_section *sections,
                              size_t secnum,
                              void *addr);
```

Description

The `dyn_SetSectionAddr` function sets the local address for a given section within a dynamically-loadable module (DLM). `image` is the DLM, validated by `dyn_ValidateImage`. `sections` is a native copy of the DLM's section table, obtained by `dyn_GetSections`. `secnum` is the number for the section for which to set the address. `addr` is the local address.

In this context, "setting the address" means informing the DLM that address addr is a suitable address at which section `secnum` may reside after relocation; if `dyn_CopySectionContents` is called, the section's contents will be copied to `addr`, so sufficient space must have previously been reserved at that address.

Error Conditions

The `dyn_SetSectionAddr` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
`DYN_NO_ERROR`	Success. The address has been recorded within the native section table copy.
`DYN_BAD_PTR`	The `sections` or `image` parameter is `NULL`, or there is no section `secnum`. This value is also returned if the section already has an address assigned, or it has already been relocated.

Example

```
#include <libdyn.h>

int set_addr(dyn_mem_image *image, dyn_section *secs,
             size_t num, void *ptr) {
   if (dyn_SetSectionAddr(image, secs, num, ptr) == DYN_NO_ERROR)
      return 0;
   return -1;
}
```

See Also

dyn_ValidateImage, dyn_GetNumSections, dyn_GetSections, dyn_GetStringTableSize, dyn_GetStringTable, dyn_GetExpSymTab, dyn_LookupByName, dyn_Relocate, dyn_AllocSectionMem, dyn_CopySectionContents

dyn_SetSectionMem

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Specify the target address of a dynamically-loadable section

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_SetSectionMem(dyn_mem_image *image,
                             dyn_section *sections,
                             size_t secnum,
                             uint32_t taddr,
                             dyn_section_mem **memptr);
```

Description

The dyn_SetSectionMem function creates internal house-keeping memory for a given section within a dynamically-loadable module (DLM), and records the target address at which the section will reside. image is the DLM, validated by dyn_ValidateImage. sections is a native copy of the DLM's section table, obtained by dyn_GetSections. secnum is the number for the section for which to set the address. taddr is the target address.

In this context, the target address refers to the address at which the section will begin, when relocated.

The function will create a dyn_section_mem structure, pointed to by *memptr, which can be passed to dyn_SetSectionAddr.

Error Conditions

The dyn_SetSectionMem function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
DYN_NO_ERROR	Success. The address has been recorded within the native section table copy.
DYN_BAD_PTR	The image, sections, or memptr parameter is NULL.
DYN_BAD_IMAGE	There is no section secnum.
DYN_NO_MEM	There is insufficient memory to allocate the internal house-keeping structures.

Example

```
#include <libdyn.h>

dyn_section_mem *set_addr(dyn_mem_image *image,  dyn_section *secs,
                          size_t num, uint32_t addr) {
   dyn_section_mem *mem = 0;
   if (dyn_SetSectionMem(image, secs, num, addr, &mem) == DYN_NO_ERROR)
      return 0;
   return mem;
}
```

See Also

dyn_ValidateImage, dyn_GetNumSections, dyn_GetSections, dyn_GetStringTableSize, dyn_GetStringTable, dyn_GetExpSymTab, dyn_LookupByName, dyn_Relocate, dyn_AllocSectionMem, dyn_CopySectionContents

dyn_ValidateImage

ATTENTION: Support for dynamically-loadable modules (DLMs) using `libdyn.h` is deprecated and to be removed in a future release. Refer to ***Dynamically-Loadable Modules*** in the System Run-Time Documentation section of help for information on how to use the adi_libldr.h APIs to load a DLM from a loader stream.

Verify a memory buffer contains a valid dynamically-loadable module

Synopsis

```
#include <libdyn.h>

DYN_RESULT dyn_ValidateImage(void *ptr,
                             size_t len,
                             dyn_mem_image *image);
```

Description

The `dyn_ValidateImage` function accepts a pointer to a block of memory, and performs various checks to determine whether the memory contains a validate dynamically-loadable module (DLM), as produced by `elf2dyn`.

The memory buffer is pointed to by `ptr`, and must be at least `len` characters in size. If the buffer does contain a valid DLM, the function will populate the structure pointed to by `image`; the resulting `image` pointer will be suitable for passing to other DLM-handling functions.

Error Conditions

The `dyn_ValidateImage` function returns a status value indicating success, or the reason for failure, as follows.

Returned Value	*Reason*
`DYN_NO_ERROR`	Success. The buffer contains a valid DLM.
`DYN_BAD_PTR`	The `ptr` or `image` parameter is `NULL`.
`DYN_TOO_SMALL`	The memory buffer as described by `ptr/len` is too small to contain any DLM, or the DLM's sections/relocations exceed the buffer.
`DYN_BAD_IMAGE`	The image does not have the right magic number, or offsets within the image are nonsensical.
`DYN_BAD_VERSION`	The DLM's version number is not a version supported by this instance of `libdyn`.
`DYN_BAD_FAMILY`	The DLM is for a processor family not recognized by this instance of `libdyn`.

Example

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

```
static dyn_mem_image dlm_info;

int check_dlm(FILE *fp, char *buf, size_t maxlen) {
   size_t len = fread(buf, 1, maxlen, fp);
   if (dyn_ValidateImage(buf, len, &dlm_info) == DYN_NO_ERROR)
      return 0;
   return -1;
}
```

See Also

dyn_ValidateImage, dyn_GetSections, dyn_GetStringTableSize, dyn_GetStringTable, dyn_GetExpSymTab, dyn_LookupByName, dyn_Relocate, dyn_SetSectionAddr, dyn_AllocSectionMem, dyn_CopySectionContents

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

None

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

None

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 being 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 if the end of file has been reached.

Error Conditions

None

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 `0`, otherwise it will return a non-zero value.

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

Error Conditions

None

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 `0`.

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 `fgetpos` 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 `NULL` 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

getc, getc, gets

fileno

Get the file descriptor for a stream

Synopsis

```
#include <stdio.h>

int fileno(FILE *stream);
```

Description

The `fileno` function returns the file descriptor for a stream. The file descriptor is an opaque value used by the extensible device driver interface to represent the open file. The resulting value may only be used as a parameter to other functions that accept file descriptors.

Error Conditions

The `fileno` function returns `-1` if it detects that `stream` is not valid or is not open. If successful, it returns a positive value.

Example

```
#include <stdio.h>

int apply_control_cmd(FILE *fp, int cmd, int val){
   int fildes = fileno(fp);
   return ioctl(fildes, cmd, val);
}
```

See Also

fopen, ioctl

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

None

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 `0`.

Error Conditions

None

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 below. If any other mode specification is selected then the behavior is undefined.

Mode	*Selection*
`r`	Open text file for reading. This operation fails if the file has not previously been created.

Mode	*Selection*
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.
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 the ***Valid Flags for fprintf Function*** table.

Table 2-38: 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 conversions `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 (see the ***Length Modifiers for fprintf Function*** table) 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 2-39: 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
ll	The argument should be interpreted as a longlong 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.

The ***Valid Conversion Specifier Definitions for fprintf Function*** table contains definitions of the valid conversion specifiers that define the type of conversion to be applied.

Table 2-40: 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

Table 2-40: Valid Conversion Specifier Definitions for fprintf Function (Continued)

Specifier	*Conversion*
`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, vsprintf, vsnprintf

fputc

Put a character on a stream

Synopsis

```
#include <stdio.h>

int fputc(int ch, FILE *stream);
```

Description

The `name ">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 `name ">fputc` function is successful then it will return the value that was written to the stream.

Error Conditions

If the `name ">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

fputc

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 `0`, then `fread` will return `0` 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 `0` 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, ferror, 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 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

None

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, heap_malloc, heap_realloc, malloc, realloc, heap_space_unused

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 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 [, 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

None

Example

```
#include <math.h>

double y;
float x;
int exponent;

y = frexp (2.0, &exponent);       /* y = 0.5, exponent = 2 */
x = frexpf (4.0, &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 find 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 (the ***Length Modifiers for fscanf Function*** table) 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 2-41: 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`.
`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.

Table 2-41: Length Modifiers for fscanf Function (Continued)

Length	*Action*
`t`	The argument should be interpreted as `ptrdiff_t`.
`z`	The argument should be interpreted as `size_t`.

NOTE: 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 the ***Valid Conversion Specifier Definitions for fscanf Function*** table.

Table 2-42: 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+6abc1.234abc234.56abc                         */
   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 as follows.

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. On SHARC processors that support byte addressing, this is 8-bit bytes; otherwise it is 32-bit words.

NOTE: 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 `fsetpos` on the same stream.

NOTE: 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

See fgetpos 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. On SHARC processors that support byte addressing, this is 8-bit bytes, otherwise it is 32-bit words.

> **NOTE:** Positioning within a file that has been opened as a text stream is only supported by the libraries that Analog Devices supplies 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 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 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

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` function has 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

None

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.

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

None

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 ***Using Multiple Heaps*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

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

Example

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

#pragma section("seg_hp2")
static char extra_heap[256];

int main()
{
   char *buf;
   int index, uid = 999; /* arbitrary userid for heap */
   /* Install extra_heap[] as a heap */
   index = heap_install(extra_heap, sizeof(extra_heap), uid);
   if (index < 0) {
```

```
      printf("installation failed\n");
      return 1;
   }

   /* Allocate memory for 128 characters from extra_heap[] */
   buf = (char *)heap_calloc(index,128,sizeof(char));
   if (buf != 0) {
      printf("Allocated space starting at %p\n", buf);
      free(buf);   /* free can be used to release the memory */
   } else {
         printf("Unable to allocate from extra_heap[]\n");
   }
    return 0;
}
```

See Also

calloc, free, heap_free, heap_malloc, heap_realloc, malloc, realloc, heap_space_unused

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.

The `heap_free` function deallocates the object whose address is `ptr`, provided that `ptr` is not a null pointer. If the object was not allocated by one of the heap allocation routines, or if the object has been previously freed, then the behavior of the function is undefined. If `ptr` is a null pointer, then the `heap_free` function will just return.

The function does not use the `heap_index` argument; instead it identifies the heap from which the object was allocated and returns the memory to this heap. For more information on creating multiple run-time heaps, see ***Using Multiple Heaps*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

None

Example

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

#pragma section("seg_hp2")
static char extra_heap[256];

int main()
{
   char *buf;
   int index, uid = 999; /* arbitrary userid for heap */
   /* Install extra_heap[] as a heap */
   index = heap_install(extra_heap, sizeof(extra_heap), uid);
   if (index < 0) {
      printf("installation failed\n");
      return 1;
   }

   /* Allocate memory for 128 characters from extra_heap[] */
   buf = (char *)heap_calloc(index,128,sizeof(char));
   if (buf != 0) {
      printf("Allocated space starting at %p\n", buf);
      heap_free(index, buf);
   } else {
      printf("Unable to allocate from extra_heap[]\n");
   }
   return 0;
}
```

See Also

calloc, free, heap_calloc, heap_malloc, heap_realloc, malloc, realloc, heap_space_unused

heap_init

Re-initialize a heap

Synopsis

```
#include <stdlib.h>

int heap_init(int heap_index);
```

Description

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

The `heap_init` function re-initializes a heap, discarding all allocations within the heap. Because the function discards any allocations within the heap, it must not be used if there are any allocations on the heap that are still active and may be used in the future.

The function returns a zero if it succeeds in re-initializing the heap specified.

NOTE: The run-time libraries use the default heap for data storage, potentially before the application has reached `main`. Therefore, re-initializing the default heap may result in erroneous or unexpected behavior.

Error Conditions

The `heap_init` function returns a non-zero result if it failed to re-initialize the heap.

Example

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

int heap_index = heap_lookup(USERID_HEAP);
if (heap_init(heap_index)!=0) {
   printf("Heap re-initialization failed\n");
}
```

See Also

calloc, free, heap_free, heap_space_unused, heap_install, heap_lookup, heap_malloc, heap_realloc, malloc, realloc, space_unused

heap_install

Sets up a heap at runtime

Synopsis

```
#include <stdlib.h>

int heap_install(void *base, size_t length, int userid);
```

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 runtime. The dynamic heap is identified by the `userid`.

Not all length words are available for user allocations. Some space is reserved for administration.

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.

Error Conditions

The `heap_install` function returns `-1` if initialization was unsuccessful. Potential reasons include: there is not enough space available in the `__heaps` table; a heap with the specified `userid` already exists; the space is not large enough for the internal heap structures.

Example

```
#include <stdlib.h>
#define EXTRAID    666
#define EXTRASZ    256

/* LDF must map this section to appropriate memory */
#pragma section("runtime_heap")
static char extra_heap[EXTRASZ];

int main()
{
   int i;
   int index;
   int *x = NULL;

   index = heap_install(extra_heap, EXTRASZ, EXTRAID);
   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_malloc

heap_lookup

Convert a `userid` to a heap index

Synopsis

```
#include <stdlib.h>

int heap_lookup(int userid);
```

Description

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

The `heap_lookup` function converts a `userid` to a heap index. All heaps have a `userid` and a heap index associated with them. Both the `userid` and the heap index are set on heap creation. The default heap has `userid 0` and heap index `0`.

The heap index is required for the functions `heap_calloc`, `heap_malloc`, `heap_realloc`, `heap_init`, and `heap_space_unused`. For more information on creating multiple run-time heaps, see ***Using Multiple Heaps*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

The `heap_lookup` function returns `-1` if there is no heap with the specified `userid`.

Example

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

int heap_userid = 1;
int heap_id;

if ( (heap_id = heap_lookup(heap_userid)) == -1) {
   printf("Lookup failed; will use the default heap\n");
   heap_id = 0;
}
char *ptr = heap_malloc(heap_id, 1024);
if (ptr == NULL) {
   printf("heap_malloc failed to allocate memory\n");
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_init, heap_install, heap_space_unused, heap_malloc, heap_realloc, malloc, realloc, space_unused

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 multiple run-time heaps, see ***Using Multiple Heaps*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

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

Example

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

#pragma section("seg_hp2")
static char extra_heap[256];

int main()
{
   char *buf;
   int index, uid = 999; /* arbitrary userid for heap */
   /* Install extra_heap[] as a heap */
   index = heap_install(extra_heap, sizeof(extra_heap), uid);
   if (index < 0) {
      printf("installation failed\n");
      return 1;
   }

   /* Allocate memory for 128 characters from extra_heap[] */
   buf = (char *)heap_malloc(index,128);
   if (buf != 0) {
      printf("Allocated space starting at %p\n", buf);
      heap_free(index, buf);
   } else {
      printf("Unable to allocate from extra_heap[]\n");
   }
```

```
    return 0;
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_realloc, malloc, realloc, heap_space_unused

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 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 re-allocated 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 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 argument `heap_index` is only used if `ptr` is a null pointer.

The memory reallocated may be deallocated with the `free` or `heap_free` function.

For more information on multiple run-time heaps, see ***Using Multiple Heaps*** in the *C/C++ Compiler Manual for SHARC Processors*.

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 <stdio.h>

#pragma section("seg_hp2")
static char extra_heap[256];
int main()
{
   char *buf, *upd;
   int index, uid = 999; /* arbitrary userid for heap */
   /* Install extra_heap[] as a heap */
   index = heap_install(extra_heap, sizeof(extra_heap), uid);
   if (index < 0) {
      printf("installation failed\n");
      return 1;
   }

   /* Allocate memory for 128 characters from extra_heap[] */
   buf = (char *)heap_malloc(index, 128);
   if (buf != 0) {
       strcpy(buf,"hello");
       /* Change allocated size to 200 */
       upd = (char *)heap_realloc(index, buf, 200);
       if (upd != 0) {
           printf("reallocated string for %s\n", upd);
           heap_free(index, upd); /* Return to extra_heap[] */
       } else {
           free(buf);   /* free can be used to release buf */
       }
   } else {
      printf("Unable to allocate from extra_heap[]\n");
   }
   return 0;
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_malloc, malloc, realloc, heap_space_unused

heap_space_unused

Space unused in specific heap

Synopsis

```
#include <stdlib.h>

int heap_space_unused(int heap_index);
```

Description

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

The `heap_space_unused` function returns the total amount of free space for the heap with index `heap_index`.

Note that calling `heap_malloc(heap_index,heap_space_unused(heap_index))` does not allocate space because each allocated block uses more memory internally than the requested space. Note also that the free space in the heap may be fragmented, and thus may not be available in one contiguous block.

Error Conditions

If a heap with heap index `heap_index` does not exist, this function returns `-1`.

Example

```
#include <stdlib.h>

int free_space;
/* Get amount of free space in heap 1 */
free_space = heap_space_unused(1);
```

See Also

calloc, free, heap_calloc, heap_free, heap_init, heap_install, heap_lookup, heap_malloc, heap_realloc, malloc, realloc, space_unused

heap_switch

Change the default heap at runtime

Synopsis

```
#include <stdlib.h>

int heap_switch (int heap_index);
```

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.

The function does not check the validity of the heap index. If the heap index is invalid, then subsequent operations on the default heap (using the functions `malloc`, `calloc`, `realloc` and `space_unused`, or using the C++ `new` operator) will return an error.

For more information on multiple run-time heaps, see ***Multiple Heaps*** in the *C/C++ Compiler Manual for SHARC Processors*.

NOTE: The `heap_switch` function is not available in multithreaded environments.

Error Conditions

None

Example

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

#define HEAP1_USERID  1
#define HEAP1_SIZE    1024

int heap1[HEAP1_SIZE];
int heap1_id;

char *pbuf;

/* Initialize */

heap1_id = heap_install (heap1, sizeof(heap1), HEAP1_USERID);

/* 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 `idivfx` family of functions computes the quotient and returns the closest integer value to the result.

Error Conditions

The `idivfx` function has 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

instrprof_request_flush

Flush the instrumented profiling data to the host

Synopsis

```
#include <instrprof.h>

void instrprof_request_flush(void);
```

Description

The `instrprof_request_flush` function attempts to flush any buffered instrumented profiling data to the host computer.

The flush occurs immediately if file I/O operations are allowed. (File I/O operations cannot be executed from interrupt handlers or from unscheduled regions in a multi-threaded application.) If the flush cannot occur immediately, it occurs the next time a profiled function is called, or returned from when file I/O operations are allowed.

NOTE: Do not include the header file `instrprof.h` or reference the function `instrprof_request_flush` in an application that is not built with instrumented profiling enabled. For more information, see ***-p*** in the *C/C++ Compiler Manual for SHARC Processors*. You can guard such code using the preprocessor macro `_INSTRUMENTED_PROFILING`. Note that the compiler only defines this macro when instrumented profiling is enabled.

Flushing data to the host is a cycle-intensive operation. Consider carefully when and where to call this function within your application. For more information, see ***Profiling With Instrumented Code*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

None

Example

```
#if defined (_INSTRUMENTED_PROFILING)
   #include <instrprof.h>
#endif

extern void do_something(void);

int main(void) {
   do_something();
   #if defined(_INSTRUMENTED_PROFILING)
      instrprof_request_flush();
   #endif
}
```

ioctl

Apply a control operation to a file descriptor

Synopsis

```
#include <stdio.h>

int ioctl(int fildes, int cmd, ...);
```

Description

The `ioctl` function applies command `cmd` to file descriptor `fildes` , along with any specified arguments for `cmd`. The file descriptor must be a value returned by invoking the `fileno` function upon some open stream `fp`.

The `ioctl` function is delegated to the device driver upon which stream `fp` was opened. The command `cmd`, and any provided arguments, are specific to the device driver; each device driver may interpret commands and arguments differently.

Error Conditions

The `ioctl` function returns `-1` if the operation is not recognized by the underlying device driver. Other return values are specific to the device driver's interpretation of the command.

Example

```
#include <stdio.h>

int apply_control_cmd(FILE *fp, int cmd, int val) {
  int fildes = fileno(fp);
  return ioctl(fildes, cmd, val);
}
```

See Also

fopen, fileno

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

None

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

None

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

None

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

None

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

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

None

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` functions are 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

None

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

None

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` functions are 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

None

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

None

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

None

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

None

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

None

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

None

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: Note that `labs (LONG_MIN) == LONG_MIN`.

Error Conditions

None

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

None

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

None

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, isalnum, 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

None

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

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 = 4.0  */
```

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 `0`, 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: Note that `llabs (LLONG_MIN)` == `LLONG_MIN`.

Error Conditions

None

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 extension to the ISO/IEC 9899:1990 C standard and the ISO/IEC 9899:1999 C standard.

The `lavg` function returns the average of the two arguments `value1` and `value2`.

Error Conditions

None

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 extension to the ISO/IEC 9899:1990 C standard and the ISO/IEC 9899:1999 C 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

None

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, 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 extension to the ISO/IEC 9899:1990 C standard and the ISO/IEC 9899:1999 C standard.

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

Error Conditions

None

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 `0`, 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 extension to the ISO/IEC 9899:1990 C standard and the ISO/IEC 9899:1999 C standard.

The `llmax` function returns the larger of its two arguments.

Error Conditions

None

Example

```
#include <stdlib.h>

long long i;
i = llmax (10LL, 8LL);    /* returns 10 */
```

See Also

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 extension to the ISO/IEC 9899:1990 C standard and the ISO/IEC 9899:1999 C standard.

The `llmin` function returns the smaller of its two arguments.

Error Conditions

None

Example

```
#include <stdlib.h>

long long i;
i = llmin (10LL, 8LL);     /* returns 8 */
```

See Also

llmax, lmax, lmin, max, min

lmax

Long int maximum

Synopsis

```
#include <stdlib.h>

long int lmax (long int value1, long int value2);
```

Description

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

None

Example

```
#include <stdlib.h>

long int i;
i = lmax (10L, 8L);     /* returns 10 */
```

See Also

llmax, 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 extension to the ISO/IEC 9899:1990 C standard and the ISO/IEC 9899:1999 C 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

None

Example

```
#include <stdlib.h>

long int i;
i = lmin (10L, 8L);    /* returns 8 */
```

See Also

llmin, lmax, 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` (see the ***Members of the lconv Struct*** table). 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 2-43: Members of the lconv Struct

Member	*Description*
char *currency_symbol	Currency symbol applicable to the locale
char *decimal_point	Used to format non-monetary quantities
char *grouping	Used to indicate the number of digits in each non monetary 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
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 non-monetary 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

None

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.hThis 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

None

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

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

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.

NOTE: 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

None

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 `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;
size_t sz = 10 * sizeof(int);

ptr = (int *)malloc (sz);     /* ptr points to an   */
                              /* array of length 10 */
```

See Also

calloc, free, heap_calloc, heap_free, heap_malloc, heap_realloc, realloc

max

Maximum

Synopsis

```
#include <stdlib.h>

int max (int value1, int value2);
```

Description

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

None

Example

```
#include <stdlib.h>

int i;
i = max (10, 8);     /* returns 10 */
```

See Also

llmax, llmin, lmax, 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

None

Example

```
#include <string.h>

char *ptr;

ptr = memchr ("TESTING", 'E', 7);
    /* ptr points to the E in TESTING */
```

See Also

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

None

Example

```
#include <string.h>

char *string1 = "ABC";
char *string2 = "BCD";
int result;

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

See Also

strcmp, strcoll, strcmp

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.

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

Error Conditions

None

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

None

Example

```
#include <string.h>

char *ptr, *str = "ABCDE";

ptr = str + 2;
memmove (ptr, str, 3);     /* ptr = "ABC", str = "ABABC" */
```

See Also

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

None

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 extension to the ISO/IEC 9899:1990 C standard and the ISO/IEC 9899:1999 C 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

None

Example

```
#include <stdlib.h>

int i;
i = min (10, 8);     /* returns 8 */
```

See Also

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

None

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 `mulifx` family of functions computes the product and returns the integer part of the result (rounding towards zero).

Error Conditions

None

Example

```
#include <stdfix.h>

unsigned long int ulprod;
ulprod = muliulr(128, 0.125ulr);        /* ulprod == 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

None

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

pgo_hw_request_flush

Request a flush to the host of the data gathered through profile-guided optimization on hardware

Synopsis

```
#include <pgo_hw.h>

void pgo_hw_request_flush(void);
```

Description

The pgo_hw_request_flush function requests that the runtime support for profile-guided optimization on hardware should write gathered data to the host computer. The flush occurs the next time the profile-guided optimization on hardware run-time support attempts to record data, as long as file I/O operations are allowed. File I/O operations cannot be executed from interrupt handlers or when in an unscheduled region in a multi-threaded application.

NOTE: Do not include the header file pgo_hw.h or reference the function pgo_hw_request_flush in an application not built for profile-guided optimization on hardware. You can guard such code using the preprocessor macro _PGO_HW; the compiler only defines this macro when profile-guided optimization for hardware is enabled.

For more information, see `-pguide` and `-prof-hw` in the *C/C++ Compiler Manual for SHARC Processors*.

Flushing data to the host is a cycle-intensive operation. Consider carefully when and where to call this function within your application. For more information, see ***Profile Guided Optimization and Code Coverage*** in the *C/C++ Compiler Manual for SHARC Processors*.

Error Conditions

None

Example

```
#if defined (_PGO_HW)
   #include <pgo_hw.h>
#endif

extern void do_something(void);

int main(void) {
   do_something();
   #if defined(_PGO_HW)
      pgo_hw_request_flush();
   #endif
}
```

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 (4.0, 2.0);      /* z = 16.0 */
x = powf (4.0, 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 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

fputc

qsort

Quick sort

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

None

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);
```

Description

The `raise` function invokes the function registered for signal `sig` by function `signal`, if any. The `sig` argument must be one of the signals listed in signal.

NOTE: The `raise` function provides the functionality described in the ISO/IEC 9899:1999 Standard, and has no impact on the processor's interrupt mechanisms. For information on handling interrupts, refer to the ***System Run-Time Documentation*** in the online help.

Error Conditions

The `raise` function returns a zero if successful or a non-zero value if `sig` is an unrecognized signal value.

Example

```
#include <signal.h>

raise(SIGABRT);     /* equivalent to calling abort() */
```

See Also

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

None

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 <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 the 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

None

Example

```
#include <21262.h>

int intmem1[100];
int intmem2[100];

/* Place extmem1 in external memory, in the used-defined */
/* section "seg_extmem"                                  */
#pragma section("seg_extmem", DMA_ONLY)
int extmem1[100];

/* Place extmem2 in external memory, in the used-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);
}
```

NOTE: 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 one of the heap functions, 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 re-allocated 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 `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_malloc, heap_realloc, heap_malloc

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 delegated to the current default device driver.

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 delegated to the current default device driver.

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

None

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 `roundfx` 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 is 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

None

Example

```
#include <stdfix.h>

long fract f;

f = roundulr(0x12345678p-32ulr, 16);   /* f == 0x12340000ulr */
f = roundfx(0x12345678p-32ulr, 16);    /* f == 0x12340000ulr */
```

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 its 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 its first argument. For details on the control format string refer to fscanf.

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.

> **NOTE:** 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

None

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

setbuf

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.

NOTE: 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

None

Example

See `longjmp` for an example.

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

None

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 the following table.

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.

NOTE: 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

signal

Define signal handling

Synopsis

```
#include <signal.h>

void (*signal (int sig, void (*func)(int))) (int);
```

Description

The `signal` function determines how to handle a signal that is triggered by the `raise` or `abort` functions. The specified function `func` can be associated with one of the `sig` values listed in the ***Valid Values for Parameter sig*** table.

NOTE: The function is not thread-safe.

Table 2-44: Valid Values for Parameter sig

Sig Value	*Meaning According to ISO/IEC 9899:1999 Standard*
`SIGTERM`	Request for program termination.
`SIGABRT`	Program is terminating abnormally.
`SIGFPE`	Arithmetic operation was erroneous, e.g. division by zero.
`SIGILL`	Illegal instruction, or equivalent.
`SIGINT`	Request for interactive attention.
`SIGSEGV`	Access to invalid memory.

NOTE: Despite the interpretations of the sig values listed in the ***Valid Values for Parameter sig*** table, the `signal` function has no effect on the processor's interrupt mechanism. Any function registered via the signal function will only be invoked if done so explicitly, via the function `abort` or the function `raise`. For information on handling processor interrupts, see the ***System Run-Time Documentation*** in the online help.

The `func` parameter may be one of the values listed in the ***Additional Valid Values for Parameter func*** table, instead of a pointer to a function.

Table 2-45: Additional Valid Values for Parameter func

func Value	*Meaning*
SIG_DFL	Default behavior: do nothing if the signal is triggered by `raise` or `abort`.
SIG_ERR	An error occurred.
SIG_IGN	Ignore the signal if triggered by `raise` or `abort`.

Return Value

The `signal` function returns the value of the previously installed signal or signal handler action.

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 (SIGABRT, abort_handler);   /* enable abort signal  */
signal (SIGABRT, SIG_IGN);         /* disable abort signal */
```

See Also

abort, 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 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

space_unused

Space unused in heap

Synopsis

```
#include <stdlib.h>

int space_unused(void);
```

Description

The `space_unused` function returns the size of the total amount of free space for the default heap. Note that calling `malloc(space_unused())` does not allocate space because each allocated block uses more memory internally than the requested space, and also the free space in the heap may be fragmented, and thus not be available in one contiguous block.

Error Conditions

If there are no heaps, calling this function will return `-1`.

Example

```
#include <stdlib.h>

int free_space;
```

```
/* Get amount of free space in the heap */
free_space = space_unused();
```

See Also

calloc, free, heap_calloc, heap_free, heap_init, heap_install, heap_lookup, heap_malloc, heap_space_unused, realloc

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 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..... */
```

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

None

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.

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

None

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

None

Example

```
#include <string.h>

char string1[50], string2[50];

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

See Also

memchr, strncmp

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

None

Example

```
#include <string.h>

char *ptr1;

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

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 run-time environment for ADSP-21xxx and ADSP-SC5xx processors, 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

None

Example

```
#include <string.h>

char string1[50], string2[50];

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

See Also

strncmp, strrchr

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

None

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

None

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

None

Example

```
#include <string.h>

char *ptr1;
ptr1 = strerror (1);
```

See Also

No related functions

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 below in the ***Conversion Specifiers Supported by strftime*** table.

Note that the `strftime` function only supports the "C" locale, and this is reflected in the table.

Table 2-46: 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
`%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

Table 2-46: Conversion Specifiers Supported by strftime (Continued)

Conversion Specifier	*Transformation*	*ISO/IEC 9899*
`%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
`%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

NOTE: 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

ctimegmtime, 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

None

Example

```
#include <string.h>

size_t len;
len = strlen ("SOMEFUN");     /* len = 7 */
```

See Also

strspn, strcspn

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

None

Example

```
#include <string.h>

char string1[50], *ptr;
string1[0] = '\0';
strncat (string1, "MOREFUN", 4);
     /* string1 equals "MORE" */
```

See Also

strcspn

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

None

Example

```
#include <string.h>

char string1[50], string2[50];

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

See Also

memchr, strncmp

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

None

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, strncpy

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

strcspn

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 length of the initial segment of `s1`, which consists entirely of characters in the string pointed to by `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

None

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 `strtofxfx` family of functions extracts a value from the string pointed to by `nptr`, and returns the value as a fixed-point. The `strtofxfx` 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 `strtofxfx` 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 functions

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 0, 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 white-space 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 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 0, 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, strtofxfx, strtoll, 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 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 0, 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 unsignedlong long 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 0, 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. It places the result in the array pointed to by `s1`.

NOTE: 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

None

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 and ADSP-SC5xx run-time environment, `system` always returns zero.

Error Conditions

None

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

None

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 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

ctimegmtime, 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

None

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

None

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

None

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

None

Example

Refer to va_arg for an example.

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

None

Example

Refer to va_arg for an example

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 for a description of the valid format specifiers.

The vfprintf function behaves in the same manner asfprintf 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 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 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 `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 `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 `NULL` 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 for a description of the valid format specifiers.

With one exception, the `vsprintf` function behaves in the same manner as `sprintf`. Instead of being a function that takes a variable number or an arguments function, 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 `NULL` 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 <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 the 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

None

Example

See read_extmem for a usage example.

See Also

read_extmem

3 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 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 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.

Similar to other functions you use, library functions should be declared. Declarations are supplied in header files, as described in Working With Library Source Code.

Note that C++ namespace prefixing is not supported when calling a DSP library function. All DSP library functions are in the C++ global namespace.

NOTE: 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, When compiling with the −`char-size-8` switch on processors that support byte-addressing, the assembly version of the function will have a dot suffix, otherwise it will have an underscore prefix. For more information on naming conventions, see ***C/C++ and Assembly Interface*** in the *C/C++ Compiler Manual for SHARC Processors*.

You can use the archiver, described in the *Linker and Utilities Manual*, to build library archive files of your own functions.

Reentrancy

All of 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 can be used safely in a multi-threaded environment (such as with an RTOS).

Library Attributes

The DSP run-time library contains the same attributes as the C/C++ run-time library. For more information, see Library Attributes.

Working With Library Source Code

The source code for functions in the C and DSP run-time libraries is provided with CCES, in the `sharc\lib\src` subdirectory.

The directory contains the source for the C run-time library, DSP run-time library, and I/O run-time library, as well as the source for the main program startup functions. Note that source code is not provided for functions that use the SHARC+ FFTA accelerator, which are contained in the `libfftacc.dlb` library.

The source code allows you to customize specific functions. To modify these files, you need proficiency in the ADSP-21xxx assembly language and an understanding of the run-time environment, as explained in ***C/C++ Run-Time Model and Environment*** in the *C/C++ Compiler Manual for SHARC Processors*.

Before modifying the source code, copy it to a file with a different filename and rename the function itself. Test the function before using it in your system to verify that it is functionally correct.

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

DSP Header Files

The DSP header files contain prototypes for all of the DSP library functions. When the appropriate `#include` preprocessor command is included in your source, the compiler uses the prototypes to check that each function is called with the correct arguments. The ***Summaries of DSP Header Files*** table provides summaries of the DSP header files supplied with this release of the `cc21k` compiler.

Table 3-1: Summaries of DSP Header Files

Header File	*Summary*
adi_fft_wrapper.h	Functions that use the SHARC+ FFTA accelerator
asm_sprt.h	Mixed C/Assembly language macros
cmatrix.h	Arithmetic between complex matrices
comm.h	Scalar companders for A-law and -law
complex.h	Basic complex arithmetic functions
cvector.h	Arithmetic between complex vectors
filter.h	Filters and transformations
filters.h	Filters operating on scalar input values
macros.h	Macros to access processor features
math.h	Math functions
matrix.h	Matrix functions
platform_include.h	Platform-specific functions
stats.h	Statistical functions
sysreg.h	Functions for access to SHARC system registers
trans.h	Fast Fourier Transform functions (not optimized for SHARC SIMD architectures)
vector.h	Vector functions
window.h	Window generators

The following sections describe the DSP header files in more detail.

adi_fft_wrapper.h

The `adi_fft_wrapper.h` header file contains function prototypes and definitions related to the use of the FFTA accelerator available on SHARC+ processors. The FFTA is an off-core hardware compute engine that may be used to perform transformations such as Fast Fourier Transforms (FFTs).

The header file defines two different sets of FFT functions. The first set comprises standalone routines which use the FFTA to perform a one-off transformation. Some of these routines provide similar functionality to functions that are declared in the `filter.h` header file. Note that these functions may impose different requirements on their parameters, as noted in the detailed descriptions for each function.

The second set of Fast Fourier Transforms allow repeated transformations to be performed on multiple datasets. This model gives greater performance, as the cost of initializing the accelerator is paid only once, and the DMA operations to send data to the accelerator and to bring results back may be overlapped, along with the computation on the accelerator itself. Most standalone functions discussed above have an equivalent function that uses this continuous model.

The FFT functions declared in this header file include versions to perform windowing of the input array, along with versions that produce magnitude squared output.

For a list of library functions available in this header, see the ***Library Functions in adi_fft_wrapper.h*** table in Documented Library Functions.

asm_sprt.h

The `asm_sprt.h` header file consists of ADSP-21xxx/SCxxx assembly language macros, not C functions. They are used in your assembly routines that interface with C functions. For more information, see ***Using Mixed C/C++ and Assembly Support Macros*** in the *C/C++ Compiler Manual for SHARC Processors*.

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 the ***Library Functions in cmatrix.h*** table in Documented Library Functions.

comm.h

The `comm.h` header file includes the voice-band compression and expansion communication functions that operate on scalar input values. For a list of library functions that use this header, see the ***Library Functions in comm.h*** table in Documented Library Functions.

> **NOTE:** Similar companding functions that operate on vectors rather than scalars are defined in the header file `filter.h`; however, the functions in `comm.h` and `filter.h` have different prototypes and therefore both header files cannot be included by the same source file. Any attempt to include both will result in the following error message being displayed:
>
> ```
> The include files comm.h and filter.h are mutually exclusive.
> Use filter.h for vectorized function arguments, or comm.h
> for scalar function arguments.
> ```

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`. It also contains definitions and prototypes for the C99 native complex support, which uses the `_Complex` keyword.

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 the ***Library Functions in complex.h*** table in Documented Library Functions.

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;
```

When C99 mode is enabled, the `complex.h` header file defines `complex` as a convenient alternative to the `_Complex` keyword, along with the unit imaginary value `I` (or `_Complex_I`). Although for convenience, some functions are provided that operate on C99 complex types, the support for C99 complex types comprises a free-standing implementation, so the full set of library functions as defined by ISO/IEC 9899 is not provided. Where the names of the C99 functions coincide with the names of functions that use the structure types, a type-generic macro is used to select the correct function based on the operand type. This mechanism, however, does not permit the address of the function to be taken. To take the address of the function, define either `ADI_COMPLEX_STRUCT_FORM` (for the structure type support) or `ADI_COMPLEX_C99` (for C99 complex functions) to avoid use of the type-generic macro.

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 the ***Library Functions in cvector.h*** table in Documented Library Functions.

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.

The header file defines three different sets of FFT functions. The first set is available only when running on SHARC SIMD processors and comprises 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, ...). These functions require the least amount of data memory space (by re-using the input array as temporary storage during execution) but at the expense of flexibility and performance. Each FFT function in this set is defined for a specific size of FFT; thus if an application calculated N different sizes of FFT, it would therefore include N different FFT library functions.

The second set of Fast Fourier Transforms are defined for all ADSP-21xxx processors and comprises the functions `cfft`, `ifft` and `rfft`. The number of points these FFT functions will compute is passed as an argument. There is also a facility to supply a twiddle table (which is a set of sine and cosine coefficients required by the FFT function) and a facility to re-use twiddle tables generated for larger FFT sizes. In addition, by explicitly supplying temporary storage, the FFT functions can be used without overwriting the input data. Compared to the first set of functions, these functions require more data memory space, but performance and code size for multiple instances is improved.

The third set of FFT functions that are 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, while they have an argument that specifies the size of the FFT, and an argument that is used to define the twiddle table, they do not have a twiddle table stride argument that allows the function to use a single table to generate different sized FFTs. Also these FFT functions overwrite the input data and the input arrays must be aligned on an address boundary that is a multiple of the FFT size. Memory usage lies between the first and second set of FFT functions.

The header file also defines library functions that compute the magnitude of an FFT, and a function that convolves two arrays.

NOTE: The header files `comm.h`, `filters.h`, and `trans.h` define functions that may have the same name as functions defined by this header file. However, the functions defined by `comm.h`,`filters.h`, and `trans.h` do not use the architecture's SIMD capabilities and they only operate on scalars. They also have different prototypes, and a source file therefore must not include `filter.h` and any of the header files `comm.h`, `filters.h`, and `trans.h`. (An error message will be generated by the header file if this situation is detected.)

For a list of library functions that use this header, see the ***Library Functions in filter.h*** table in Documented Library Functions.

filters.h

The `filters.h` header file includes a finite impulse response filter, an infinite impulse response filter, and a biquad function. These functions do not use the architecture's SIMD capabilities and only operate on scalars. For a list of library functions that use this header, see the ***Library Functions in filter.h*** table in Documented Library Functions.

NOTE: An alternative set of filter functions is defined by the header file `filters.h`; these functions use the same names and operate on vectors instead of scalars. However, they have different parameters and so a source file cannot include both header files; any attempt to include both will result in the following error message being displayed:

```
These include files filters.h and filter.h are mutually exclusive.

Use filter.h for vectorized function arguments, or filters.h for scalar
function arguments.
```

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 CCES tool set. 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.

The ***Math Library - Additional Functions*** table provides a summary of the additional library functions defined by the `math.h` header file.

Table 3-2: 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);`

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);`

Table 3-2: Math Library - Additional Functions (Continued)

Description	*Prototype*

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 the ***Library Functions in math.h*** table in Documented Library Functions.

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 the ***Library Functions in matrix.h*** table in Documented Library Functions.

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. `platform_include.h` causes one or two 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
- Header Files That Allow Access to Memory-Mapped Registers From C/C++ Code

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. The ***Header Files for Processor-Specific Register Bits*** table provides the header file names for processor-specific register bits.

Table 3-3: Header Files for Processor-Specific Register Bits

Header File	*Processor*
`def21160.h`	ADSP-21160 bit definitions
`def21161.h`	ADSP-21161 bit definitions
`def21261.h`	ADSP-21261 bit definitions

Table 3-3: Header Files for Processor-Specific Register Bits (Continued)

Header File	*Processor*
def21262.h	ADSP-21262 bit definitions
def21266.h	ADSP-21266 bit definitions
def21362.h	ADSP-21362 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
def21467.h	ADSP-21467 bit definitions
def21469.h	ADSP-21469 bit definitions
def21477.h	ADSP-21477 bit definitions
def21478.h	ADSP-21478 bit definitions
def21479.h	ADSP-21479 bit definitions
def21483.h	ADSP-21483 bit definitions
def21486.h	ADSP-21486 bit definitions
def21487.h	ADSP-21487 bit definitions
def21488.h	ADSP-21488 bit definitions
def21489.h	ADSP-21489 bit definitions
def21584.h	ADSP-21584 bit definitions
def21587.h	ADSP-21587 bit definitions
defSC582.h	ADSP-SC582 bit definitions
defSC584.h	ADSP-SC584 bit definitions
defSC587.h	ADSP-SC587 bit definitions
defSC589.h	ADSP-SC589 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;
```

NOTE: Use this method of accessing memory-mapped registers in preference to using `asm` statements.

Supplied header files are:

Cdef21160.h	Cdef21161.h	Cdef21261.h	Cdef21262.h
Cdef21266.h	Cdef21362.h	Cdef21363.h	Cdef21364.h
Cdef21365.h	Cdef21366.h	Cdef21367.h	Cdef21368.h
Cdef21369.h	Cdef21371.h	Cdef21375.h	Cdef21467.h
Cdef21469.h	Cdef21477.h	Cdef21478.h	Cdef21479.h
Cdef21483.h	Cdef21486.h	Cdef21487.h	Cdef21488.h
Cdef21489.h	cdef21584.h	cdef21587.h	cdefSC582.h
cdefSC584.h	cdefSC587.h	cdefSC589.h	

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 the ***Library Functions in stats.h*** table in Documented Library Functions.

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 ***Access to System Registers*** in the *C/C++ Compiler Manual for SHARC Processors*.

trans.h

The `trans.h` header file defines a set of Fast Fourier Transform (FFT) functions that operate on data in which the real and imaginary parts of both the input and output signal are stored in separate vectors. The functions that are defined by the header file include a set of functions that compute a complex FFT, the inverse of a complex FFT, and a set of functions that compute an FFT using real data only (this is equivalent to computing an FFT whose imaginary input component is set to zero).

Each function in this header file uses a built-in twiddle table and is designed to handle a specific size of FFT. For example, the function `cfft32` computes a complex FFT with 32 data points, `ifft64` computes the inverse of a complex FFT that has 64 data points, and `rfft128` computes a real FFT with 128 data points. The sizes of FFT supported are 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 and 65536. For a list of library functions that use this header, see the ***Supported Library Functions in trans.h*** table in Documented Library Functions.

NOTE: None of the functions defined here use the SHARC SIMD capability. However, an alternative set of functions with the same names are defined in the header file `filter.h`, but these functions have different parameters and their prototypes are incompatible. For this reason, a source file cannot include both these header files, and any attempt to do so will result in the following error message being displayed:

```
The include files trans.h and filter.h are mutually exclusive. Use filter.h for
function arguments of type complex_float, or trans.h otherwise.
```

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 the ***Library Functions in vector.h*** table in Documented Library Functions.

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 the ***Window Generator Functions*** table.

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 3-4: 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 the ***Supported Library Functions in window.h*** table in Documented Library Functions.

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 the `builtins.h` header file and have names which begin with double underscores, `__builtin`.

The built-in DSP library functions supported by the cc21k compiler are listed in the ***Built-in DSP Functions*** table. Refer to Using Compiler Built-In C Library Functions for more information.

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

Table 3-5: Built-in DSP Functions

`avg`	`clip`	`copysign`	`copysignf`
`favg`	`favgf`	`fmax`	`fmaxf`
`fmin`	`fminf`	`labs`	`lavg`
`lclip`	`lmax`	`lmin`	`max`
`min`			

NOTE: Functions `copysign`, `favg`, `fmax`, and `fmin` are compiled as a built-in function only if `double` is the same size as `float`.

If you want to use the C run-time library functions of the same name instead of the built-in function, refer to ***builtins.h*** in the *C/C++ Compiler Manual for SHARC Processors*.

Implications of Using SIMD Mode

All SHARC processors supported by CCES can perform SIMD (Single-Instruction, Multiple-Data) operations which can double the computational rate over the normal SISD (Single-Instruction, Single-Data) operations; the increase in performance occurs because memory accesses and computations are performed in pairs using the architecture's second processing element. See ***A Brief Introduction to SIMD Mode*** in the *C/C++ Compiler Manual for SHARC Processors*, which explains the mode in more detail.

The DSP run-time library makes extensive use of the processors' SIMD capabilities. However different SHARC processors have different constraints regarding memory access in SIMD mode and you should refer to the appropriate hardware reference manual regarding the restrictions that apply to your processor. As an example, SIMD memory accesses using the ADSP-2116x family of processors should be double-word aligned, and for some processors SIMD access to external memory is not supported (see Using Data in External Memory for more information on this topic). Because of these restrictions, it is important to ensure that all vectors and arrays that are passed as arguments to the DSP library functions can be accessed safely in SIMD mode.

Alternative versions of the DSP run-time library functions will be linked into an application if the corresponding C source that calls the DSP library function is compiled with the switch `-no-simd`. The alternative versions that will be included in an application will use normal SISD operations and not the processor's SIMD mode. Refer to ***SIMD Support*** in the *C/C++ Compiler Manual for SHARC Processors* for more information about how the compiler uses the SIMD feature. Some DSP library functions do not use SIMD due to the nature of their algorithm and so are not affected by any constraints associated with the feature. These library functions include all `long double` functions and the window generators. Other functions that do not use SIMD mode are listed in the ***Functions Not Using the SIMD Feature*** table.

Table 3-6: Functions Not Using the SIMD Feature

biquad	cmatmmlt	cmatsmlt	convolve
cvecdot	cvecsmlt	fir_decima	fir_interp
iir	histogram	matmmlt	matinv
transpm	zero_cross		

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.

All data passed to the FFT functions `cfftf`, `ifftf`, and `rfftf_2`must be allocated in internal memory. There are no versions of these functions that support data in external memory.

Using the SHARC+ FFTA Accelerator

This section explains how to use the FFTA accelerator available on SHARC+ processors. It contains the following topics:

- Modes of Operation
- Using Single-Shot Functions
- Continuous and Pipelined Operation
- Data Buffer Alignment
- DMA Transfer Size
- Data Placement

Modes of Operation

The library for the SHARC+ FFTA accelerator supports two modes of operation. The simplest is the single-shot mode of operation, where a single function is called, which uses the accelerator to perform a one-off data transformation. This mode of operation is described in Using Single-Shot Functions.

The second mode of operation is used when you wish to perform repeated transformations of the same type on multiple data sets. This programming model is more complex but permits you to achieve greatly increased performance by saving on repeated accelerator set-up costs and pipelining your transformations to achieve maximum throughput. This mode of operation is described in Continuous and Pipelined Operation.

Most functionality is available in both modes of operation. However, interleaved FFT/inverse FFT operations are only available in the continuous mode of operation.

Using Single-Shot Functions

The simplest way to use the SHARC+ FFTA accelerator is using the single-shot mode of operation, where a single function is called, which uses the accelerator to perform a one-off data transformation. Functions using this method begin with `accel_` and do not contain either `pipe` or `intleav` in their names. For example, accel_cfft_small is a single-shot FFTA function. These functions are contained in the library `libfftacc.dlb`. If you intend to perform transformations of the same type on multiple data sets, it is more efficient to use the accelerator in the Continuous and Pipelined Operation mode.

A function which uses the single-shot mode of operation should only be called when the FFTA is not already being used. When called, the function initializes the accelerator, programs it for the operation requested, and transfers the given data to the accelerator. When the accelerator has performed the operation, the function transfers the results back from the accelerator, and closes the accelerator down so that it is again free for use.

If an error occurs when using the single shot function (such as a buffer of incorrect alignment being passed to the function) the function returns `NULL`. You should therefore check that the value returned is not NULL to be sure that no errors have occurred. Should an error have occurred, you can include the following line in your source code before the call, to obtain more information about the error:

```
accel_fft_set_error_handler(accel_fft_error_handler_print);
```

See accel_fft_set_error_handler for more information.

Guidelines that can help you get the best performance from the Single-Shot functions are discussed in Data Buffer Alignment.

Continuous and Pipelined Operation

The SHARC+ FFTA accelerator can be used to efficiently perform the same transformation on multiple data sets using the Continuous mode of operation. Functions that initialize the accelerator to use this method are those which begin with `accel_` and contain either `pipe` or `intleav` in their names. For example, accel_cfft_small_pipe is a continuous FFTA initialization function. These functions are contained in the library `libfftacc.dlb`. If you only want to perform a transformation on a single data set, it is easier to use the accelerator in the Using Single-Shot Functions mode.

To use the FFTA in continuous mode, it must first be initialized by calling the initialization function discussed above. The function's first parameter is a pointer to free memory which is used to store information on the state of the FFT device driver. Your application code is responsible for allocating this memory, and freeing it if necessary after the FFT device is closed. The memory allocated must be of size `ADI_FFT_MEMORY`, and can easily be created by declaring a variable of type `ADI_FFT_DEVICE_MEMORY` whose address is passed to the initialization function. This function returns a handle of type `ADI_FFT_HANDLE` which is used in all subsequent interactions with the accelerator.

If an error occurs when using the accelerator initialization functions (such as a buffer of incorrect alignment being passed to the function) the function will return `NULL`. You should therefore check that the handle returned is not NULL to be sure that no errors have occurred. Should an error have occurred, you can include the following line in your source code before the call, to obtain more information about the error:

```
accel_fft_set_error_handler(accel_fft_error_handler_print);
```

See accel_fft_set_error_handler for more information.

After initialization, data is passed to the FFTA using DMA. This can be achieved in two different ways, using either the FFT device driver APIs, or by programming a DMA descriptor chain manually. The first of these is simpler, while the latter offers more flexibility and performance. These approaches are covered in the following topics:

- Using the FFT device driver APIs to program DMA transfers (small FFTs only), described in Programming DMA With FFT Driver APIs.
- Programming DMA descriptor chains manually for small FFTs, described in Programming DMA Manually: Small FFTs.
- Using large FFTs in the continuous operation mode, described in Using Large FFTs in Continuous Mode.
- Using the FFTA to perform a block FIR filter, described in Performing a FIR Filter using the FFTA accelerator.

Once all desired accelerator transformations have been performed, the FFT device should be closed by calling the `adi_fft_Close` function.

Additional guidelines that can help you to get the best performance from the Continuous mode of operation are discussed in DMA Transfer Size and Data Placement.

Programming DMA With FFT Driver APIs

The easiest way to transfer data to and from the FFTA is by calling the `adi_fft_Write` and `adi_fft_Read` interfaces provided by the FFT device driver. These are blocking interfaces, and so only return when the DMA transfer is complete. To allow concurrent use of the core whilst the DMA transfers are occurring, you can use the `adi_fft_SubmitTxBuffer` and `adi_fft_GetTxBuffer` interfaces (for sending data to the accelerator), and the `adi_fft_SubmitRxBuffer` and `adi_fft_GetRxBuffer` interfaces (for receiving data from the accelerator). The `SubmitBuffer` interfaces initiate the DMA transfers, but return immediately. A subsequent call to the `GetBuffer` interfaces waits until the DMA transfer is complete, or returns immediately if it has already completed. The following shows how these interfaces are called:

```
   /* Submit num_samples of data from data_buffer */
adi_fft_SubmitTxBuffer(handle, data_buffer, num_samples);
   /* Enable DMA channel. Only do once, after first SubmitBuffer. */
adi_fft_EnableTx(handle, true);
   /* Core may do work here. */
   /* Ensure DMA transfer finished. p points
      to data buffer whose transfer is complete. */
adi_fft_GetTxBuffer(handle, &p);
```

NOTE: The third argument to SubmitBuffer functions specifies the number of units of the DMA transfer size that are to be transmitted by the DMA. The accelerator initialization functions set the DMA transfer size for both send and receive channels to 8 bytes. Therefore, unless the DMA transfer size is changed using the `adi_fft_SetTxDmaTransferSize` or `adi_fft_SetRxDmaTransferSize` functions, then all buffers should be aligned to 8-byte boundaries and the number of samples should be specified in units of 8 bytes.

Up to two buffers may be sent (or received) at any one time, allowing each DMA channel to work on two buffers in a ping-pong fashion. Each call to `GetBuffer` returns when the earlier of the two transfers completes.

The FFTA is capable of performing DMA transfers on both the receive and transmit channels concurrently with computation occurring on accelerator itself. To achieve this, the DMA and computation operations must be pipelined. For example, the data for data set N may be transferred by DMA to the accelerator, while concurrently the accelerator computes an FFT on data set N-1, and DMA transfers the results of the transformation on data set N-2 back from the accelerator to memory. To achieve this, you must send the data set N (using `accel_fft_SubmitTxBuffer`) before performing the synchronization step (by calling `adi_fft_GetRxBuffer`) for the results of an FFT on a previous data set N-2. Examples that show how this may be done are given in the Board Support Package for ADSP-SC58x processors, and may be found in CCES through Browse Examples in the Help menu.

Additional guidelines that can help you to get the best performance from the Continuous mode of operation are discussed in DMA Transfer Size and Data Placement.

Programming DMA Manually: Small FFTs

While using the FFT driver APIs is the easiest way to program the DMA transfers when using the FFTA in continuous, pipelined operation, it is possible to program the DMA manually by creating DMA descriptor chains. Interaction and synchronization of the DMA with the processor core may be achieved by programming the DMA to raise interrupts and/or use the Trigger Routing Unit (TRU). This technique involves careful design of the DMA descriptor chain in order to achieve maximum concurrent use of the memory resources. A typical technique is to use ping-pong buffering, where the descriptor chain is set to alternate between two buffers, but in some cases it may be desirable to use more than two buffers for maximum performance.

The DMA descriptor chain can often be set up statically in your program. This involves declaring a linked list of TCB entries, of type `ADI_DMA_TCB[3-7]`, declared in <adi_dma.h>. This type contains the values that will be loaded into the DMA registers. Macros, `ACCEL_FFT_{RT}X_CONFIG` are provided for ease of generating values for the `Config` field, and are defined in <adi_fft_wrapper.h>. Typical code is given below:

```
static ADI_DMA_TCB3 tcb_in[] = {
  {&tcb_in[1], in_array1,
   ACCEL_FFT_TX_CONFIG(/*2d*/false, /*trigwait*/true, /*xtrigout*/false,
                       /*ytrigout*/false, /*xinterrupt*/true,
                       /*yinterrupt*/false, LOGMSIZE, /*nexttcbsize*/3u) },
  {&tcb_in[0], in_array2,
   ACCEL_FFT_TX_CONFIG(/*2d*/false, /*trigwait*/true, /*xtrigout*/false,
                       /*ytrigout*/false, /*xinterrupt*/true,
                       /*yinterrupt*/false, LOGMSIZE, /*nexttcbsize*/3u) },
};
```

Note that the `trigwait` macro parameter specifies whether the DMA will wait for a trigger before loading the subsequent DMA descriptor.

After defining your DMA descriptor chain, and setting up the FFT for continuous, pipelined operation, the next step is typically to initialize the TRU for use.

```
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS) {
   printf("Failed to initialize TRU\n");
}
adi_fft_RegisterTxTriggerMaster(h, TRGM_SYS_SOFT0_MST);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
```

Here both the RX and TX DMA channels have been set up to listen for software-generated triggers.

The next step is to load the TX and RX DMA descriptor chains into the DMA registers. This is achieved by calling one of the `adi_fft_Submit{RT}xDescriptorChain[3-7]` functions. The size suffix (3 to 7) of the function called should match the size of the first TCB to be loaded. Additional initial parameters for the other DMA configuration registers must be specified. In addition, the final parameter specifies whether the DMA should wait for a TRU trigger before loading and starting the first DMA transfer. For example, for the TX channel you might do:

```
adi_fft_SubmitTxDescriptorChain3(h, tcb_out, XCount, XModify,
                                 YCount, YModify, /*wait*/true)
```

The `adi_fft_Submit{RT}xDescriptorChain` family of functions take care of flushing the DMA descriptors (but not the buffers they reference) from cache where necessary so it is not necessary to perform this step explicitly in your application. Likewise, when called on SHARC+ cores, it also translates any L1 addresses, both of the descriptors and the buffers they reference, to their equivalent system space alias, as required for use by the DDE. Therefore it is also unnecessary to perform such translation explicit in your application.

The DMA channels can then be enabled so that the transfers start (if they are not waiting for a TRU signal). E.g.

```
adi_fft_EnableTx(h, true)
```

Next you need to create the control loop in your program. This will be responsible for synchronization steps, setting up input data for the FFTA and processing the results, along with any other tasks you wish to perform while waiting for the FFTA to complete. A simple loop might be the following:

```
while (1) {
  bool ready;
  adi_fft_IsSynchronizeTxReady(h, &ready);
  if (ready) {
    adi_fft_SynchronizeTx(h);
    prepare_data();
    adi_fft_TriggerTx(h);
  } else {
    adi_fft_IsSynchronizeRxReady(h, &ready);
    if (ready) {
      adi_fft_SynchronizeRx(h);
      process_results();
      adi_fft_TriggerRx(h);
    } else {
      do_work_while_waiting();
    }
  }
}
```

It is also possible to register your own handlers for the TX and RX DMA interrupts by using the `adi_fft_Register{RT}xCallback` APIs. This allows you to take a callback when a DMA completes instead of using the `adi_fft_Synchronize{RT}x` routines used above.

NOTE: When using the FFTA in this way, you are responsible for any cache flushing that needs to be performed. If your input data buffer resides in cacheable memory, then after modifying the buffer it should be flushed from memory using the `flush_data_buffer` API in the `ADI_FLUSH_DATA_NOINV` mode. This prevents the DMA transfer from accessing stale data.

If your results buffer resides in cacheable memory, then before triggering the RX DMA channel to permit transferring new results, the buffer should be flushed using the `flush_data_buffer` API in the `ADI_FLUSH_DATA_NOINV` mode. The data should then not be accessed until the RX DMA channel says the next set of results are ready. When the results are ready, the buffer should be invalidated by calling the `flush_data_buffer` API in the `ADI_FLUSH_DATA_INV` mode.

The FFTA is capable of performing DMA transfers on both the receive and transmit channels concurrently with computation occurring on accelerator itself. To achieve this, the DMA and computation operations must be pipelined. For example, the data for data set N may be transferred by DMA to the accelerator, while concurrently the accelerator computes an FFT on data set N-1, and DMA transfers the results of the transformation on data set N-2 back from the accelerator to memory. To achieve this, you must send the data set N before performing the synchronization step (e.g. by calling `adi_fft_SynchronizeRx`) for the results of an FFT on a previous data set N-2.

Additional guidelines that can help you to get the best performance from the Continuous mode of operation are discussed in DMA Transfer Size and Data Placement.

Using Large FFTs in Continuous Mode

Large FFTs can also be used in non-blocking, continuous repeated operation. To do this, you need to create a DMA descriptor chain which will perform the large FFT, using one of the `accel_{cir}fft_large_set_tcbs` family of functions.

```
uint32_t tcbs[ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(NUM_POINTS,
                                /* twid_stride*/1u) / sizeof(uint32_t)];
accel_cfft_large_set_tcbs(h, tcbs, in_array, out_array, twiddles,
                                /* twid_stride*/1, NUM_POINTS);
```

This DMA descriptor chain is set up to repeatedly send the same input buffer and receive results into the same output buffer. The DMA transfers wait for triggers (using the Trigger Routing Unit, or TRU) to synchronize with the core, so that new input data is only read when the core signals that it is in place, and the result buffer is only overwritten when the core signals that the previous set of results have been processed.

After defining your DMA descriptor chain, and setting up the FFT for continuous, repeated operation, the next step is to initialize the TRU for use. The TX DMA channel must be set to listen for a trigger generated by the RX DMA channel:

```
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS) {
   printf("Failed to initialize TRU\n");
}
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
```

The DMA channels can then be enabled so that the transfers start (and wait for the first TRU signal).

```
adi_fft_EnableTx(h, true);
```

Next you need to create the control loop in your program. This will be responsible for synchronization steps, setting up input data for the FFTA and processing the results, along with any other tasks you wish to perform while waiting for the FFTA to complete. A simple loop might be the following:

```
while (1) {
   bool ready;
   adi_fft_IsSynchronizeTxReady(h, &ready);
   if (ready) {
      adi_fft_SynchronizeTx(h);
       prepare_data();
       adi_fft_TriggerTx(h);
   } else {
      adi_fft_IsSynchronizeRxReady(h, &ready);
      if (ready) {
         adi_fft_SynchronizeRx(h);
         process_results();
         adi_fft_TriggerRx(h);
      } else {
         do_work_while_waiting();
```

```
        }
    }
}
```

It is also possible to register your own handlers for the TX and RX DMA interrupts by using the `adi_fft_Register{RT}xCallback` APIs. This allows you to take a callback when a DMA completes instead of using the `adi_fft_Synchronize{RT}x` routines used above.

While using the FFTA to perform large FFTs, no pipelining is possible at the application level (the DMA descriptor chain already specifies to perform as many parallel transfers as possible). Therefore only one input and output buffer are used. When the interrupt is raised by the TX DMA channel (or it is determined to have finished using the `adi_fft_SynchronizeTx` function), the data is ready to be overwritten and, when done, the TX channel can be triggered. When the RX DMA channel raises an interrupt, or it is determined to have finished using the `adi_fft_SynchronizeRx` function, the results can be processed and once done, the RX channel can be trigged to allow more results to be received.

NOTE: When using the FFTA in this way, you are responsible for any cache flushing that needs to be performed. If your input data buffer resides in cacheable memory, then after modifying the buffer it should be flushed from memory using the `flush_data_buffer` API in the `ADI_FLUSH_DATA_NOINV` mode. This prevents the DMA transfer from accessing stale data.

If your results buffer resides in cacheable memory, then before triggering the RX DMA channel to permit transferring new results, the buffer should be flushed using the `flush_data_buffer` API in the `ADI_FLUSH_DATA_NOINV` mode. The data should then not be accessed until the RX DMA channel says the next set of results are ready. When the results are ready, the buffer should be invalidated by calling the `flush_data_buffer` API in the `ADI_FLUSH_DATA_INV` mode.

Additional guidelines that can help you to get the best performance from the Continuous mode of operation are discussed Data Placement. For best performance, the input, output and any temp buffer should be aligned to 32 bytes.

Performing a FIR Filter using the FFTA accelerator

The FFTA accelerator can also be used for perform a block FIR filter. This performs convolution of the input data and the FIR coefficients in the frequency domain, before performing an inverse FFT to return output in the time domain. Your application is responsible for providing the filter coefficients in the frequency domain and for performing an overlap-and-add on the output data, as well as managing the delay line. Information on how to do this is given below, and an example application is provided in the Board Support Package.

First of all, your FIR filter coefficients must be transformed into the frequency domain so that they can be consumed by the FFTA algorithm. This is achieved by performing the following:

- Split your `N_TAPS` array of real coefficients into chunks of `BLOCK_SIZE`.
- Append `BLOCK_SIZE` zeros to the end of each `BLOCK_SIZE` chunk of coefficients to produce chunks of `(2*BLOCK_SIZE)` real elements.

- Perform a real FFT on each (2*BLOCK_SIZE) chunk to produce output of (2*BLOCK_SIZE) complex elements.
- Concatentate all the (2*BLOCK_SIZE) output arrays together to produce an array of (2*N_TAPS) complex coefficients that are used as input to the FIR filter algorithm.

Note that this transformation of the filter coefficients can be performed externally to your application and the transformed coefficients then linked statically into your program.

Next, you need to create a DMA descriptor chain which will be responsible for performing the DMA transfers to the FFTA, using the `accel_fft_fir_set_tcbs` function.

```
uint32_t tcbs[ACCEL_FFT_FIR_TCB_CHAIN_SIZE(BLOCK_SIZE, N_TAPS) /
                       sizeof(uint32_t)];
accel_fft_fir_set_tcbs(h, tcbs, in_array, out_array, coeffs,
                       BLOCK_SIZE, N_TAPS);
```

This DMA descriptor chain is set up to repeatedly send the same input buffer and receive results into the same output buffer. The DMA transfers wait for triggers (using the Trigger Routing Unit, or TRU) to synchronize with the core, so that new input data is only read when the core signals that it is in place, and the result buffer is only overwritten when the core signals that the previous set of results have been processed.

After defining your DMA descriptor chain, and setting up the FFT for repeated FIR filtering using the `accel_fft_fir_pipe` function, the next step is to initialize the TRU for use. The TX DMA channel must be set to listen for a trigger generated by the RX DMA channel:

```
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS) {
   printf("Failed to initialize TRU\n");
}
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
```

The DMA channels can then be enabled so that the transfers start (and wait for the first TRU signal).

```
adi_fft_EnableTx(h, true);
```

Next you need to create the control loop in your program. This will be responsible for synchronization steps, setting up input data for the FIR filter and processing the results, along with any other tasks you wish to perform while waiting for the FFTA to complete. In addition, your control loop needs to perform an overlap-and-add on the output data, and must maintain the delay line. A simple loop might be the following:

```
while (1) {
   bool ready;
   copy_input_data(in_array); /* copy latest data into input array */
   adi_fft_TriggerTx(h);
   adi_fft_TriggerRx(h);
   adi_fft_IsSynchronizeTxReady(h, &ready);
   while (!ready) {
      do_work_while_waiting();
      adi_fft_IsSynchronizeTxReady(h, &ready);
   }
   adi_fft_SynchronizeTx(h);
```

```
    adi_fft_IsSynchronizeRxReady(h, &ready);
    while (!ready) {
       do_work_while_waiting();
       adi_fft_IsSynchronizeRxReady(h, &ready);
    }
    adi_fft_SynchronizeRx(h);
    /* overlap and add */
    delayIndex = 0u;
    for (i = 0u; i < (N_TAPS / BLOCK_SIZE); i++) {
       uint32_t j;
       for (j = 0u; j < 2u * BLOCK_SIZE; j++) {
          delay[delayIndex] += out_array[j + (i * 2u * BLOCK_SIZE)];
          delayIndex++;
       }
       delayIndex -= BLOCK_SIZE;
    }
    process_results(delay); /* results now at bottom of delay line */
    /* Shift the delay line */
    for (i = 0u; i < N_TAPS; i++) {
       delay[i] = delay[BLOCK_SIZE + i];
    }
    for (i = 0u; i < BLOCK_SIZE; i++) {
       delay[N_TAPS + i] = 0.0f;
    }
}
```

It is also possible to register your own handlers for the TX and RX DMA interrupts by using the `adi_fft_Register{RT}xCallback` APIs. This allows you to take a callback when a DMA completes instead of using the `adi_fft_Synchronize{RT}x` routines used above.

While using the FFTA to perform a FIR filter, no pipelining is possible at the application level (the DMA descriptor chain already specifies to perform as many parallel transfers as possible). Therefore only one input and output buffer are used. When the interrupt is raised by the TX DMA channel (or it is determined to have finished using the `adi_fft_SynchronizeTx` function), the data is ready to be overwritten and, when done, the TX channel can be triggered. When the RX DMA channel raises an interrupt, or it is determined to have finished using the `adi_fft_SynchronizeRx` function, the results can be processed and once done, the RX channel can be trigged to allow more results to be received.

NOTE: When using the FFTA in this way, you are responsible for any cache flushing that needs to be performed. If your input data buffer resides in cacheable memory, then after modifying the buffer it should be flushed from memory using the `flush_data_buffer` API in the `ADI_FLUSH_DATA_NOINV` mode. This prevents the DMA transfer from accessing stale data.

If your results buffer resides in cacheable memory, then before triggering the RX DMA channel to permit transferring new results, the buffer should be flushed using the `flush_data_buffer` API in the `ADI_FLUSH_DATA_NOINV` mode. The data should then not be accessed until the RX DMA channel says the next set of results are ready. When the results are ready, the buffer should be invalidated by calling the `flush_data_buffer` API in the `ADI_FLUSH_DATA_INV` mode.

Additional guidelines that can help you to get the best performance from the Continuous mode of operation are discussed Data Placement. The FIR algorithm accesses the coefficients array and the output array simultaneously, so these should be placed in different memory banks where possible. For best performance, the input, output and coefficients buffer should be aligned to 32 bytes.

Data Buffer Alignment

There are several constraints or recommendations regarding the alignment of data buffers when using the SHARC+ FFTA Accelerator.

- Certain functions impose alignment constraints on the data buffer parameters passed to them. These constraints are documented in the description of each function. In particular, all large FFTs (i.e. where the number of points exceeds 2048) require any input or output buffer of complex data to be aligned to at least an 8 byte boundary. If a buffer which does not meet this requirement is passed to the function, an error occurs and the function will return `NULL`.
- DMA transfers are faster when a larger DMA transfer size is used. For the Continuous mode of operation, see DMA Transfer Size. When using Single-Shot functions, a larger DMA transfer size (resulting in faster execution) will be used if the function is passed aligned input and ouput buffers, up to alignment on a 32-byte boundary.
- The initialization functions for Continuous mode with windowing will also execute faster if the window array is aligned more strictly.
- When an output buffer is held in cacheable memory, the buffer must be aligned to a cache-line boundary. This is to prevent incorrect operation in the case that nearby accesses cause parts of the output buffer to be brought into cache. If an output buffer which does not meet this requirement is passed to a Single-Shot function, an error occurs and the function returns `NULL`.

NOTE: The alignment of a data array can be specified using `#pragma align N`.

DMA Transfer Size

When the FFTA accelerator is used with the continuous mode of operation, the DMA transfer size may be set for both the send and receive channels. When using the FFT driver routines to perform the DMA transfers, this can be changed by using the `adi_fft_SetTxDmaTransferSize` and `adi_fft_SetRxDmaTransferSize` functions. Alternatively, if programming the DMA descriptor chain manually, the transfer size is specified as a parameter to the `ADI_FFT_{RT}X_CONFIG` macro.The transfer sizes supported are 4, 8, 16 and 32 bytes. On initialization of the accelerator, the transfer size for both channels is set to 8 bytes.

Best performance is achieved by using the maximum transfer size possible. However the transfer size imposes the following constraints:

- The data buffer from which data is sent to the accelerator, or the data buffer into which data is received, must be aligned to the DMA transfer size for the DMA channel, and be contiguous in memory. For example, if the data being sent resides in a buffer which is aligned to a 16 byte boundary, you can use `adi_fft_SetTxDmaTransferSize` to `ADI_FFT_DMA_TRANSFER_16BYTES` to achieve better performance.

- When using the `adi_fft_SubmitTxBuffer`, `adi_fft_SubmitRxBuffer`, `adi_fft_Write` and `adi_fft_Read` interfaces to send or receive data, the number of samples being sent is specified in units of the DMA transfer size for the DMA channel. Therefore, if for an N-point complex FFT you change the DMA transfer size to 32 bytes, you should change the number of samples being sent from N to N/4.

NOTE: The alignment of a data array can be specified using `#pragma align N`.

Data Placement

To achieve maximum performance when the FFTA Accelerator is used in the continuous mode of operation, the DMA transfer for transmitting data to the accelerator should occur concurrently with the DMA transfer for receiving data from the accelerator. For these two DMA channels to access memory in parallel, the data buffers being accessed in parallel should:

- Be located in distinct blocks in L1 memory
- Use different slave ports to L1 memory

Locating the buffers in distinct blocks in L1 memory can be achieved by using `#pragma section("section_name")` to control data placement. The use of different slave ports is performed automatically by the FFT device driver, and no code changes are required in your application to achieve this.

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.

The ***Library Functions in adi_fft_wrapper.h*** table lists the library functions in the `adi_fft_wrapper.h` header file. Refer to adi_fft_wrapper.h for more information on this header file.

Table 3-7: Library Functions in adi_fft_wrapper.h

accel_cfft	accel_cfftN	accel_cfft_intleav_small
accel_cfft_intleav_small_mag_sq	accel_cfft_intleav_small_windowed	accel_cfft_intleav_small_windowed_mag_sq
accel_cfft_large	accel_cfft_large_mag_sq	accel_cfft_large_mag_sq_pipe
accel_cfft_large_mag_sq_set_tcbs	accel_cfft_large_mag_sq_set_tcbs_strided	accel_cfft_large_pipe
accel_cfft_large_set_tcbs	accel_cfft_large_set_tcbs_strided	accel_cfft_large_windowed

Table 3-7: Library Functions in adi_fft_wrapper.h (Continued)

accel_cfft_large_windowed_mag_sq	accel_cfft_large_windowed_mag_sq_pipe	accel_cfft_large_windowed_mag_sq_set_tcbs
accel_cfft_large_windowed_mag_sq_set_tcbs_strided	accel_cfft_large_windowed_pipe	accel_cfft_large_windowed_set_tcbs
accel_cfft_large_windowed_set_tcbs_strided	accel_cfft_small	accel_cfft_small_mag_sq
accel_cfft_small_mag_sq_pipe	accel_cfft_small_pipe	accel_cfft_small_windowed
accel_cfft_small_windowed_mag_sq	accel_cfft_small_windowed_mag_sq_pipe	accel_cfft_small_windowed_pipe
accel_fft_fir_multi_channel_set_tcbs	accel_fft_fir_pipe	accel_fft_fir_set_tcbs
accel_fft_set_error_handler	accel_ifft	accel_ifftN
accel_ifft_large	accel_ifft_large_pipe	accel_ifft_large_set_tcbs
accel_ifft_large_set_tcbs_strided	accel_ifft_large_windowed	accel_ifft_large_windowed_pipe
accel_ifft_large_windowed_set_tcbs	accel_ifft_large_windowed_set_tcbs_strided	accel_ifft_small
accel_ifft_small_pipe	accel_ifft_small_windowed	accel_ifft_small_windowed_pipe
accel_irfft_large	accel_irfft_large_pipe	accel_irfft_large_set_tcbs
accel_irfft_large_set_tcbs_strided	accel_irfft_large_windowed	accel_irfft_large_windowed_pipe
accel_irfft_large_windowed_set_tcbs	accel_irfft_large_windowed_set_tcbs_strided	accel_irfft_small
accel_irfft_small_pipe	accel_irfft_small_windowed	accel_irfft_small_windowed_pipe
accel_rfft	accel_rfftN	accel_rfft_intleav_small
accel_rfft_intleav_small_mag_sq	accel_rfft_intleav_small_windowed	accel_rfft_intleav_small_windowed_mag_sq
accel_rfft_large	accel_rfft_large_mag_sq	accel_rfft_large_mag_sq_pipe
accel_rfft_large_mag_sq_set_tcbs	accel_rfft_large_mag_sq_set_tcbs_strided	accel_rfft_large_pipe
accel_rfft_large_set_tcbs	accel_rfft_large_set_tcbs_strided	accel_rfft_large_windowed
accel_rfft_large_windowed_mag_sq_pipe	accel_rfft_large_windowed_mag_sq_set_tcbs	accel_rfft_large_windowed_mag_sq_set_tcbs
accel_rfft_large_windowed_pipe	accel_rfft_large_windowed_set_tcbs	accel_rfft_large_windowed_set_tcbs_strided
accel_rfft_large_windowed_mag_sq	accel_rfft_small	accel_rfft_small_mag_sq
accel_rfft_small_mag_sq_pipe	accel_rfft_small_pipe	accel_rfft_small_windowed
accel_rfft_small_windowed_mag_sq	accel_rfft_small_windowed_mag_sq_pipe	accel_rfft_small_windowed_pipe
accel_twidfft		

The ***Library Functions in cmatrix.h*** table lists the library functions in the `cmatrix.h` header file. Refer to cmatrix.h for more information on this header file.

Table 3-8: Library Functions in cmatrix.h

cmatmadd	cmatmmlt	cmatmsub

Table 3-8: Library Functions in cmatrix.h (Continued)

cmatsadd	cmatsmlt	cmatssub

The ***Library Functions in comm.h*** table lists the library functions in the `comm.h` header file. Refer to comm.h for more information on this header file

Table 3-9: Library Functions in comm.h

a_compress	a_expand	mu_compress
mu_expand		

The ***Library Functions in complex.h*** table lists the library functions in the `complex.h` header file. Refer to complex.h for more information on this header file.

Table 3-10: Library Functions in complex.h

arg	cabs	cadd
cartesian	cdiv	cexp
cmlt	conj	csub
norm	polar	

The ***Library Functions in cvector.h*** table lists the library functions in the `cvector.h` header file. Refer to cvector.h for more information on this header file.

Table 3-11: Library Functions in cvector.h

cvecdot	cvecsadd	cvecsmlt
cvecssub	cvecvadd	cvecvmlt
cvecvsub		

The ***Library Functions in filter.h*** table lists the library functions in the `filter.h` header file. Refer to filter.h for more information on this header file.

Table 3-12: Library Functions in filter.h

a_compress	a_expand	biquad
biquad_trans	cfft	cfft_mag
cfftN	cfftf	convolve
fft_magnitude	fftf_magnitude	fir
fir_decima	fir_interp	firf
ifft	ifftf	ifftN
iir	mu_compress	mu_expand
rfft	rfft_mag	rfftf_2

Table 3-12: Library Functions in filter.h (Continued)

rfftN	twidfft	twidfftf

The ***Library Functions in math.h*** table lists the library functions in the `math.h` header file. Refer to math.h for more information on this header file.

Table 3-13: Library Functions in math.h

alog	alog10	copysign
cot	favg	fclip
fmax	fmin	rsqrt

The ***Library Functions in matrix.h*** table lists the library functions in the `matrix.h` header file. Refer to matrix.h for more information on this header file.

Table 3-14: Library Functions in matrix.h

matinv	matmadd	matmmlt
matmsub	matmadd	matsmlt
matssub	transpm	

The ***Library Functions in stats.h*** table lists the library functions in the `stats.h` header file. Refer to stats.h for more information on this header file.

Table 3-15: Library Functions in stats.h

autocoh	autocorr	crosscoh
crosscorr	histogram	mean
rms	var	zero_cross

The ***Library Functions in trans.h*** table lists the library functions in the `trans.h` header file. Refer to trans.h for more information on this header file.

Table 3-16: Library Functions in trans.h

cfftN	ifftN	rfftN

The ***Library Functions in vector.h*** table lists the library functions in the `vector.h` header file. Refer to vector.h for more information on this header file.

Table 3-17: Library Functions in vector.h

vecdot	vecsadd	vecsmlt
vecssub	vecvadd	vecvmlt
vecvsub		

The ***Library Functions in window.h*** table lists the library functions in the `window.h` header file. Refer to window.h for more information on this header file.

Table 3-18: 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.

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* - 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

accel_cfft

Complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_cfft (const complex_float  dm input[],
                                  complex_float  dm output[],
                            const complex_float  pm twiddle[],
                                  int               twiddle_stride,
                                  float             scale,
                                  int               n);
```

Description

The accel_cfft function uses the SHARC+ FFTA accelerator to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the scale parameter. The value of n, which represents the number of points in the FFT, must be a power of 2 between 64 and 4194304. If you know that n is less than or equal to 2048, it is better to use the accel_cfft_small function; if n is known to be greater than 2048 it is better to use the accel_cfft_large function. Due to the overheads associated with setting up the FFTA accelerator for one-time use, it is not recommended to use this function for FFTs of fewer than 1024 points, as it may give poorer performance than computing the FFT on the SHARC+ core.

The size of the input array input and the output array output must be at least n. The output array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The other arrays must all be aligned to at least an 8 byte boundary. Best performance is achieved if they are aligned to at least a 32 byte boundary.

The twiddle table is only used for FFTs of 4096 points or more. The minimum size of the twiddle table is n/2 if generated by the twidfft function. A larger twiddle table may be used, provided that the value of the twiddle table stride argument twiddle_stride is set to the FFT size corresponding to the twiddle table, divided by n.

Twiddle tables accel_twiddles_N, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the libfftacc.dlb library. It is most efficient to use these twiddle tables which are optimized for use by the FFTA. Alternatively, the library function twidfft 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 returns the address of the output array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32

complex_float output[N_FFT];
int twiddle_stride = 1;
   /* Compute Fast Fourier Transform */
complex_float *result = accel_cfft(input, output,
                                   accel_twiddles_4096, twiddle_stride,
```

```
                                   1.0, N_FFT);
    if (!result)
       return 1; /* error! */
```

See Also

accel_cfft_large, accel_cfft_small, accel_cfft_small_pipe, accel_fft_set_error_handler, twidfft

accel_cfftN

Complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_cfftN (const complex_float  dm input[],
                             complex_float        dm output[]);
```

Description

The `accel_cfftN` function, where N is one of 1024, 2048, 4096, 8192, 16384, 32768 or 65536, uses the SHARC+ FFTA accelerator to transform 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` and the output array `output` must be at least N. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. Best performance is achieved if arrays are aligned to at least a 32 byte boundary.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>

#pragma align 32
complex_float  input[1024];
#pragma align 32

complex_float  output[1024];
```

```
   /* Compute Fast Fourier Transform */
complex_float *result = accel_cfft1024(input, output);
if (!result)
   return 1; /* error! */
```

See Also

accel_cfft, accel_cfft_large, accel_cfft_small_pipe, accel_fft_set_error_handler

accel_cfft_intleav_small

Set-up function for interleaved complex and inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_cfft_intleav_small (ADI_FFT_DEVICE_MEMORY *h,
                                         float                 cscale,
                                         float                 iscale,
                                         int                   n);
```

Description

The `accel_cfft_intleav_small` function sets up the SHARC+ FFTA accelerator to perform pipelined alternate transformations of the time domain complex input signal sequence to the frequency domain, and inverse transformations of the frequency domain complex input signal sequence to the time domain, by using the radix-2 Fast Fourier Transform (FFT). The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The second and third parameters, `cscale` and `iscale`, specify the scaling factors used for the forward and inverse transformations respectively. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the all input and output data transfers must be `n` complex data points. While the results of the forward and inverse FFTs may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets (one for the forward, and one for the inverse transformation) in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  2048

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_cfft_intleav_small(&mem, 1.0, 1.0/N_FFT, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_cfft_intleav_small_mag_sq, accel_cfft_intleav_small_windowed, accel_cfft_intleav_small_windowed_mag_sq, accel_cfft_small, accel_cfft_small_pipe, accel_ifft_small, accel_ifft_small_pipe, accel_rfft_intleav_small, accel_fft_set_error_handler

accel_cfft_intleav_small_mag_sq

Set-up function for interleaved magnitude-squared complex and inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_cfft_intleav_small_mag_sq (ADI_FFT_DEVICE_MEMORY *h,
                                                float                 cscale,
                                                float                 iscale,
                                                int                   n);
```

Description

The `accel_cfft_intleav_small_mag_sq` function sets up the SHARC+ FFTA accelerator to perform pipelined alternate transformations of the time domain complex input signal sequence to the frequency domain, and inverse transformations of the frequency domain complex input signal sequence to the time domain, by using the radix-2 Fast Fourier Transform (FFT). The forward transformation computes the magnitude squared of each element of the output. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The second and third parameters, `cscale` and `iscale`, specify the scaling factors used for the forward and inverse transformations respectively. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the output data transfers of the forward transformations must be `n` real data points; all other input and output data transfers must be `n` complex data points. While the results of the forward and inverse FFTs may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets (one for the forward, and one for the inverse transformation) in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  2048

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_cfft_intleav_small_mag_sq(&mem, 1.0, 1.0/N_FFT, N_FFT);
if (!h)
   return 1; /* error! */
```

```
   /* DMA transfers to and from accelerator here */
```

See Also

accel_cfft_intleav_small, accel_cfft_intleav_small_windowed, accel_cfft_intleav_small_windowed_mag_sq, accel_cfft_small_mag_sq, accel_cfft_small_mag_sq_pipe, accel_ifft_small, accel_ifft_small_pipe, accel_rfft_intleav_small_mag_sq, accel_fft_set_error_handler

accel_cfft_intleav_small_windowed

Set-up function for interleaved windowed complex and inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_cfft_intleav_small_windowed (ADI_FFT_DEVICE_MEMORY *h,
                                                  const float           *window,
                                                  float                 cscale,
                                                  float                 iscale,
                                                  int                   n);
```

Description

The `accel_cfft_intleav_small_windowed` function sets up the SHARC+ FFTA accelerator to perform pipelined alternate transformations of the time domain complex input signal sequence to the frequency domain, and inverse transformations of the frequency domain complex input signal sequence to the time domain, by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data to the forward transformation by multiplying the input data by the `window` array. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The third and fourth parameters, `cscale` and `iscale`, specify the scaling factors used for the forward and inverse transformations respectively. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the all input and output data transfers must be `n` complex data points. While the results of the forward and inverse FFTs may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets (one for the forward, and one for the inverse transformation) in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes..

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  2048

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
#pragma align 32
float window[N_FFT];
   /* Set up accelerator for Fast Fourier Transform */
h = accel_cfft_intleav_small_windowed(&mem, window, 1.0, 1.0/N_FFT, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_cfft_intleav_small, accel_cfft_intleav_small_mag_sq, accel_cfft_intleav_small_windowed_mag_sq, accel_cfft_small, accel_cfft_small_pipe, accel_ifft_small, accel_ifft_small_pipe, accel_rfft_intleav_small, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_cfft_intleav_small_windowed_mag_sq

Set-up function for interleaved windowed magnitude-squared complex and inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_cfft_intleav_small_windowed_mag_sq(
                                    ADI_FFT_DEVICE_MEMORY *h,
```

```
                                    const float           *window,
                                    float                 cscale,
                                    float                 iscale,
                                    int                   n);
```

Description

The accel_cfft_intleav_small_windowed_mag_sq function sets up the SHARC+ FFTA accelerator to perform pipelined alternate transformations of the time domain complex input signal sequence to the frequency domain, and inverse transformations of the frequency domain complex input signal sequence to the time domain, by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data to the forward transformation by multiplying the input data by the window array, and the magnitude squared is computed for each element of the output. The value of n, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, h, to the function must point to free memory which is used for storing the state of the FFT device. The third and fourth parameters, cscale and iscale, specify the scaling factors used for the forward and inverse transformations respectively. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the output data transfers of the forward transformations must be n real data points; all other input and output data transfers must be n complex data points. While the results of the forward and inverse FFTs may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets (one for the forward, and one for the inverse transformation) in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  2048
```

```
ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
#pragma align 32
float window[N_FFT];
   /* Set up accelerator for Fast Fourier Transform */
h = accel_cfft_intleav_small_windowed_mag_sq(&mem, window, 1.0, 1.0/N_FFT, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_cfft_intleav_small, accel_cfft_intleav_small_mag_sq, accel_cfft_intleav_small_windowed, accel_cfft_small_pipe, accel_ifft_small, accel_ifft_small_pipe, accel_rfft_intleav_small_windowed_mag_sq, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_cfft_large

Complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_cfft_large (const complex_float  dm input[],
                                  complex_float        dm output[],
                                  const complex_float  pm twiddle[],
                                  int                  twiddle_stride,
                                  float                scale,
                                  int                  n);
```

Description

The `accel_cfft_large` function uses the SHARC+ FFTA accelerator to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The size of the input array `input` and the output array `output` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is n. A larger twiddle table may be used, provided that the value of the twiddle table stride argument twiddle_stride is set to the FFT size corresponding to the twiddle table, divided by n.

Twiddle tables accel_twiddles_N, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the libfftacc.dlb library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA function.

The function returns the address of the output array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096
#pragma align 32

complex_float input[N_FFT];
#pragma align 32
complex_float output[N_FFT];
int twiddle_stride = 1;
   /* Compute Fast Fourier Transform */
complex_float *result = accel_cfft_large(input, output,
                                         accel_twiddles_4096, twiddle_stride,
                                         1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_cfft, accel_cfft_small, accel_cfft_small_pipe, accel_fft_set_error_handler, accel_twidfft

accel_cfft_large_mag_sq

Complex radix-2 Fast Fourier Transform producing magnitude-squared output, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_cfft_large_mag_sq (const complex_float  dm input[],
                                               float          dm output[],
                                               complex_float  dm temp[],
                                         const complex_float  pm twiddle[],
                                               int            twiddle_stride,
                                               float          scale,
                                               int            n);
```

Description

The `accel_cfft_large_mag_sq` function uses the SHARC+ FFTA accelerator to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The size of the input array `input` array, the output array `output` and the `temp` array must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary. The `temp` array is used for temporary storage during the execution of the function.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA function.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
float output[N_FFT];
#pragma align 32
complex_float temp[N_FFT];
int twiddle_stride = 1;
   /* Compute Fast Fourier Transform */
complex_float *result = accel_cfft_large_mag_sq
                              (input, output, temp,
                               accel_twiddles_4096, twiddle_stride,
                               1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_cfft_large, accel_cfft_small, accel_cfft_small_mag_sq_pipe, accel_fft_set_error_handler, accel_twidfft

accel_cfft_large_mag_sq_pipe

Set-up function for large (`n` >= 4096) complex radix-2 Fast Fourier Transform producing magnitude-squared output, using the SHARC+ FFTA accelerator in non-blocking mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_cfft_large_mag_sq_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                             float                 scale,
                                             int                   n);
```

Description

The `accel_cfft_large_mag_sq_pipe` function sets up the SHARC+ FFTA accelerator to perform non-blocking transformations of the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, a DMA descriptor chain must be created to handle the DMA transfers to and from the accelerator. This is done by calling the accel_cfft_large_mag_sq_set_tcbs function. The requirements on data buffers, such as any alignment constraints, are specified in the documentation of that function. For more information on how to perform large FFTs in non-blocking mode, see Using the SHARC+ FFTA Accelerator, and in particular Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_cfft_large_mag_sq_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */

 /* Set up DMA descriptor chain here */
```

See Also

accel_cfft_large_mag_sq, accel_cfft_large_mag_sq_set_tcbs, accel_cfft_large_mag_sq_set_tcbs_strided, accel_fft_set_error_handler

accel_cfft_large_mag_sq_set_tcbs

Set up DMA descriptor chain to perform DMA transfers for large (`n` >= 4096 points) complex radix-2 Fast Fourier Transform producing magnitude-squared output, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_cfft_large_mag_sq_set_tcbs (ADI_FFT_HANDLE       h,
                                         void                *tcb,
                                         const complex_float  dm input[],
                                         float                dm output[],
                                         complex_float        dm temp[],
                                         const complex_float  pm twiddle[],
```

```
                              int                  twiddle_stride,
                              int                  n);
```

Description

The `accel_cfft_large_set_tcbs` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_cfft_large_mag_sq_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_MAG_SQ_TCB_CHAIN_SIZE(n, twid_stride).`

The size of the input array `input`, the output array `output` and the `temp` array must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise they must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary. The `temp` array is used for temporary storage during computation of the FFT.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by `accel_cfft_large_mag_sq_set_tcbs` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
float  output[N_FFT];
#pragma align 32
complex_float  temp[N_FFT];
int    twiddle_stride = 1;
uint32_t  tcbmem[ACCEL_FFT_LARGE_MAG_SQ_TCB_CHAIN_SIZE(N_FFT, 1u) /
                                       sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
    /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_cfft_large_mag_sq_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
 /* Create DMA descriptor chain */
void *tcb = accel_cfft_large_mag_sq_set_tcbs
                              (h, tcbmem, input,
                               output, temp, accel_twiddles_4096,
                               twiddle_stride, N_FFT);
if (!tcb)
    return 1; /* error! */
    /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
    return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
    /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
    /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_cfft_large_mag_sq_pipe, accel_cfft_large_mag_sq_set_tcbs_strided, accel_fft_set_error_handler, accel_twidfft

accel_cfft_large_mag_sq_set_tcbs_strided

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) complex radix-2 Fast Fourier Transform producing magnitude-squared output, using the SHARC+ FFTA accelerator, permitting both input and output arrays to be non-contiguous in memory.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_cfft_large_mag_sq_set_tcbs_strided
                                   (ADI_FFT_HANDLE       h,
                                    void                *tcb,
                                    const complex_float  dm input[],
                                    int                  input_stride,
                                    float                dm output[],
                                    int                  output_stride,
                                    complex_float        dm temp[],
                                    const complex_float  pm twiddle[],
                                    int                  twiddle_stride,
                                    int                  n);
```

Description

The accel_cfft_large_set_tcbs_strided function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. The value of n, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_cfft_large_mag_sq_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro ACCEL_FFT_LARGE_MAG_SQ_TCB_CHAIN_SIZE(n, twid_stride).

The input array input and the output array output must comprise at least n(strided) data elements. The size of the temp array must be at least n. The output and temp arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise they must be aligned to at least an 8 byte boundary. The twiddle array must be aligned to at least a 32 byte boundary. The input array must be aligned to at least an 8 byte boundary. The strides, input_stride and output_stride, specify the stride in data element units; a stride of 1 indicates that the data is contiguous in memory. Best performance is achieved if all contiguous arrays are aligned to at least a 32 byte boundary. The temp array is used for temporary storage during computation of the FFT.

The minimum size of the twiddle table is n. A larger twiddle table may be used, provided that the value of the twiddle table stride argument twiddle_stride is set to the FFT size corresponding to the twiddle table, divided by n.

Twiddle tables accel_twiddles_N, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the libfftacc.dlb library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by accel_cfft_large_mag_sq_set_tcbs_strided contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both adi_fft_SubmitTxDescriptorChain and adi_fft_SubmitRxDescriptorChain before returning. To subsequently enable the DMA, it is only necessary to call adi_fft_EnableTx and adi_fft_EnableRx. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

complex_float input[2 * N_FFT];
#pragma align 32
float  output[256 * N_FFT];
#pragma align 32
complex_float  temp[N_FFT];
int    twiddle_stride = 1;
int    input_stride = 2;
int    output_stride = 256;
uint32_t  tcbmem[ACCEL_FFT_LARGE_MAG_SQ_TCB_CHAIN_SIZE(N_FFT, 1u) /
                                       sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
    /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_cfft_large_mag_sq_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
 /* Create DMA descriptor chain */
void *tcb = accel_cfft_large_mag_sq_set_tcbs_strided
                              (h, tcbmem, input, input_stride,
```

```
                              output, output_stride, temp, accel_twiddles_4096,
                              twiddle_stride, N_FFT);
if (!tcb)
    return 1; /* error! */
    /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
    return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
    /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
    /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_cfft_large_mag_sq_pipe, accel_cfft_large_mag_sq_set_tcbs, accel_fft_set_error_handler, accel_twidfft

accel_cfft_large_pipe

Set-up function for large (`n` >= 4096) complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator in non-blocking mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_cfft_large_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                      float                 scale,
                                      int                   n);
```

Description

The `accel_cfft_large_pipe` function sets up the SHARC+ FFTA accelerator to perform non-blocking transformations of the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, a DMA descriptor chain must be created to handle the DMA transfers to and from the accelerator. This is done by calling the accel_cfft_large_set_tcbs function. The requirements on data buffers, such as any alignment constraints, are specified in the documentation of that function. For more information on how to perform large FFTs in non-blocking mode, see Using the SHARC+ FFTA Accelerator, and in particular Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_cfft_large_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */

 /* Set up DMA descriptor chain here */
```

See Also

accel_cfft_large, accel_cfft_large_set_tcbs, accel_cfft_large_set_tcbs_strided, accel_fft_set_error_handler

accel_cfft_large_set_tcbs

Set up DMA descriptor chain to perform DMA transfers for large (`n` >= 4096 points) complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_cfft_large_set_tcbs (ADI_FFT_HANDLE       h,
                                  void                *tcb,
                                  const complex_float  dm input[],
                                  complex_float        dm output[],
                                  const complex_float  pm twiddle[],
                                  int                  twiddle_stride,
                                  int                  n);
```

Description

The `accel_cfft_large_set_tcbs` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_cfft_large_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(n, twid_stride).`

The size of the input array `input` and the output array `output` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by `accel_cfft_large_set_tcbs` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
complex_float  output[N_FFT];
int twiddle_stride = 1;
uint32_t tcbmem[ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(N_FFT, 1u) / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_cfft_large_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_cfft_large_set_tcbs (h, tcbmem, input,
                                       output, accel_twiddles_4096,
                                       twiddle_stride, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_cfft_large_pipe, accel_cfft_large_set_tcbs_strided, accel_fft_set_error_handler, accel_twidfft

accel_cfft_large_set_tcbs_strided

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator, permitting both input and output arrays to be non-contiguous in memory.

Synopsis

```
#include <adi_fft_wrapper.h>
```

```
void  *accel_cfft_large_set_tcbs_strided (ADI_FFT_HANDLE       h,
                                          void                *tcb,
                                          const complex_float  dm input[],
                                          int                  input_stride,
                                          complex_float        dm output[],
                                          int                  output_stride,
                                          complex_float        dm temp[],
                                          const complex_float  pm twiddle[],
                                          int                  twiddle_stride,
                                          int                  n);
```

Description

The `accel_cfft_large_set_tcbs_strided` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_cfft_large_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(n, twid_stride).`

The input array `input` and the output array `output` must comprise at least `n`(strided) data elements. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise they must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. The strides, `input_stride` and `output_stride`, specify the stride in data element units; a stride of 1 indicates that the data is contiguous in memory. Best performance is achieved if all contiguous arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The `temp` parameter may be used to provide temporary storage which can give improved performance when the value of `output_stride` is greater than 1. If used, it must point to an array of at least `n` elements. If not used, the value `NULL` may be passed in, in which case the `output` array itself is used for intermediate stages of the FFT computation.

The TX DMA descriptor chain created by `accel_cfft_large_set_tcbs_strided` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

complex_float input[2 * N_FFT];
#pragma align 32
complex_float output[256 * N_FFT];
#pragma align 32
complex_float temp[N_FFT];
int twiddle_stride = 1;
int input_stride = 2;
int output_stride = 256;
uint32_t tcbmem[ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(N_FFT, 1u) / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_cfft_large_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_cfft_large_set_tcbs_strided (h, tcbmem, input, input_stride,
                                               output, output_stride, temp,
                                               accel_twiddles_4096,
                                               twiddle_stride, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
```

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_cfft_large_pipe, accel_cfft_large_set_tcbs, accel_fft_set_error_handler, accel_twidfft

accel_cfft_large_windowed

Complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator, with windowing.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_cfft_large_windowed (const complex_float  dm input[],
                                           complex_float        dm output[],
                                           const complex_float  pm twiddle[],
                                           int                  twiddle_stride,
                                           const float          dm window[],
                                           float                scale,
                                           int                  n);
```

Description

The `accel_cfft_large_windowed` function uses the SHARC+ FFTA accelerator to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The results are scaled by the value of the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The size of the input array `input`, the output array `output` and the window array `window` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA function.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
complex_float output[N_FFT];
#pragma align 32
float window[N_FFT];
int twiddle_stride = 1;
   /* Compute Fast Fourier Transform */
complex_float *result = accel_cfft_large_windowed
                                 (input, output,
                                  accel_twiddles_4096, twiddle_stride,
                                  window, 1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_cfft_large, accel_cfft_small_windowed, accel_cfft_small_windowed_pipe, accel_fft_set_error_handler, accel_twidfft, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_cfft_large_windowed_mag_sq

Complex radix-2 Fast Fourier Transform producing magnitude-squared output, using the SHARC+ FFTA accelerator, with windowing.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float *accel_cfft_large_windowed_mag_sq (const complex_float dm input[],
                                                 float               dm output[],
                                                 complex_float       dm temp[],
                                                 const complex_float pm twiddle[],
                                                 int   widdle_stride,
                                                 const float dm window[],
                                                 float scale,
                                                 int   n);
```

Description

The `accel_cfft_large_windowed_mag_sq` function uses the SHARC+ FFTA accelerator to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The results are scaled by the value of the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The size of the input array `input`, the output array `output`, the window array `window` and the `temp` array must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The twiddle array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary. The `temp` array is used for temporary storage during execution of the function.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA function.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
float output[N_FFT];
#pragma align 32
complex_float temp[N_FFT];
#pragma align 32
float window[N_FFT];
int   twiddle_stride = 1;
   /* Compute Fast Fourier Transform */
complex_float *result = accel_cfft_large_windowed_mag_sq
                                 (input, output,
                                  temp, accel_twiddles_4096,
                                  twiddle_stride, window,
                                  1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_cfft_large_windowed, accel_cfft_small_windowed_mag_sq, accel_cfft_small_windowed_mag_sq_pipe, accel_fft_set_error_handler, accel_twidfft, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_cfft_large_windowed_mag_sq_pipe

Set-up function for large (`n` >= 4096) complex radix-2 Fast Fourier Transform with windowing and producing magnitude-squared output, using the SHARC+ FFTA accelerator in non-blocking mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_cfft_large_windowed_mag_sq_pipe
                                       (ADI_FFT_DEVICE_MEMORY *h,
                                        float                 scale,
                                        int                   n);
```

Description

The `accel_cfft_large_windowed_mag_sq_pipe` function sets up the SHARC+ FFTA accelerator to perform non-blocking transformations of the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), with windowing, and computes the magnitude-squared of each element of the output. The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, a DMA descriptor chain must be created to handle the DMA transfers to and from the accelerator. This is done by calling the accel_cfft_large_windowed_set_tcbs function. The requirements on data buffers, such as any alignment constraints, are specified in the documentation of that function. For more information on how to perform large FFTs in non-blocking mode, see Using the SHARC+ FFTA Accelerator, and in particular Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  4096

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_cfft_large_windowed_mag_sq_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */

 /* Set up DMA descriptor chain here */
```

See Also

accel_cfft_large_windowed_mag_sq, accel_cfft_large_windowed_mag_sq_set_tcbs, accel_cfft_large_windowed_mag_sq_set_tcbs_strided, accel_fft_set_error_handler

accel_cfft_large_windowed_mag_sq_set_tcbs

Set up DMA descriptor chain to perform DMA transfers for large (`n >= 4096` points) complex radix-2 Fast Fourier Transform with windowing and producing magnitude-squared output, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>
void  *accel_cfft_large_windowed_mag_sq_set_tcbs
                                (ADI_FFT_HANDLE       h,
                                 void                 *tcb,
                                 const complex_float dm input[],
                                 float                dm output[],
                                 complex_float        dm temp[],
                                 const complex_float pm twiddle[],
                                 int                  twiddle_stride,
                                 const float          dm window[],
                                 int                  n);
```

Description

The accel_cfft_large_windowed_mag_sq_set_tcbs function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. A window function is applied to the input data prior to the FFT transformation by multiplying the input array by the window array. The value of n, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_cfft_large_windowed_mag_sq_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro ACCEL_FFT_LARGE_WINDOWED_MAG_SQ_TCB_CHAIN_SIZE(n, twid_stride).

The size of the input array input, the window array, the output array output and the temp array must be at least n. The output and temp arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise they must be aligned to at least an 8 byte boundary. The twiddle array must be aligned to at least a 32 byte boundary. The input array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary. The temp array is used for temporary storage during computation of the FFT.

The minimum size of the twiddle table is n. A larger twiddle table may be used, provided that the value of the twiddle table stride argument twiddle_stride is set to the FFT size corresponding to the twiddle table, divided by n.

Twiddle tables accel_twiddles_N, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the libfftacc.dlb library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by `accel_cfft_large_windowed_mag_sq_set_tcbs` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT  4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
float output[N_FFT];
#pragma align 32
float window[N_FFT];
#pragma align 32
complex_float temp[N_FFT];
int twiddle_stride = 1;
uint32_t tcbmem[ACCEL_FFT_LARGE_WINDOWED_MAG_SQ_TCB_CHAIN_SIZE(N_FFT, 1u)
             / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_cfft_large_windowed_mag_sq_pipe(&mem, 1.0, N_FFT);

if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_cfft_large_windowed_mag_sq_set_tcbs
                              (h, tcbmem, input,
                               output, temp,
                               accel_twiddles_4096, twiddle_stride,
                               window, N_FFT);
if (!tcb)
```

```
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_cfft_large_windowed_mag_sq_pipe, accel_cfft_large_windowed_mag_sq_set_tcbs_strided, accel_fft_set_error_handler, accel_twidfft

accel_cfft_large_windowed_mag_sq_set_tcbs_strided

Set up DMA descriptor chain to perform DMA transfers for large (`n` >= 4096 points) complex radix-2 Fast Fourier Transform with windowing and producing magnitude-squared output, using the SHARC+ FFTA accelerator, permitting both input and output arrays to be non-contiguous in memory.

Synopsis

```
#include <adi_fft_wrapper.h>
void  *accel_cfft_large_windowed_mag_sq_set_tcbs_strided
                                (ADI_FFT_HANDLE      h,
                                 void                *tcb,
                                 const complex_float dm input[],
                                 int                 input_stride,
                                 float               dm output[],
                                 int                 output_stride,
                                 complex_float       dm temp[],
                                 const complex_float pm twiddle[],
                                 int                 twiddle_stride,
                                 const float         dm window[],
                                 int                 n);
```

Description

The `accel_cfft_large_windowed_mag_sq_set_tcbs_strided` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_cfft_large_windowed_mag_sq_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_WINDOWED_MAG_SQ_TCB_CHAIN_SIZE(n, twid_stride)`.

The input array `input` and the output array `output` must comprise at least `n`(strided) data elements. The size of the `window` array and the `temp` array must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise they must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. The strides, `input_stride` and `output_stride`, specify the stride in data element units; a stride of 1 indicates that the data is contiguous in memory. Best performance is achieved if all contiguous arrays are aligned to at least a 32 byte boundary. The `temp` array is used for temporary storage during computation of the FFT.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by `accel_cfft_large_windowed_mag_sq_set_tcbs_strided` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT  4096

complex_float input[2 * N_FFT];
#pragma align 32
float output[256 * N_FFT];
#pragma align 32
float window[N_FFT];
#pragma align 32
complex_float temp[N_FFT];
int twiddle_stride = 1;
int input_stride = 2;
int output_stride = 256;
uint32_t tcbmem[ACCEL_FFT_LARGE_WINDOWED_MAG_SQ_TCB_CHAIN_SIZE(N_FFT, 1u)
             / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_cfft_large_windowed_mag_sq_pipe(&mem, 1.0, N_FFT);

if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_cfft_large_windowed_mag_sq_set_tcbs_strided
                              (h, tcbmem, input, input_stride,
                               output, output_stride, temp,
                               accel_twiddles_4096, twiddle_stride,
                               window, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_cfft_large_windowed_mag_sq_pipe, accel_cfft_large_windowed_mag_sq_set_tcbs, accel_fft_set_error_handler, accel_twidfft

accel_cfft_large_windowed_pipe

Set-up function for large (n >= 4096) complex radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator in non-blocking mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_cfft_large_windowed_pipe (ADI_FFT_DEVICE_MEMORY  *h,
                                               float                  scale,
                                               int                    n);
```

Description

The `accel_cfft_large_windowed_pipe` function sets up the SHARC+ FFTA accelerator to perform non-blocking transformations of the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), with windowing. The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, a DMA descriptor chain must be created to handle the DMA transfers to and from the accelerator. This is done by calling the accel_cfft_large_windowed_set_tcbs function. The requirements on data buffers, such as any alignment constraints, are specified in the documentation of that function. For more information on how to perform large FFTs in non-blocking mode, see Using the SHARC+ FFTA Accelerator, and in particular Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_cfft_large_windowed_pipe(&mem, 1.0, N_FFT);
```

```
if (!h)
   return 1; /* error! */

 /* Set up DMA descriptor chain here */
```

See Also

accel_cfft_large_windowed, accel_cfft_large_windowed_set_tcbs, accel_cfft_large_windowed_set_tcbs_strided, accel_fft_set_error_handler

accel_cfft_large_windowed_set_tcbs

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) complex radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>
void  *accel_cfft_large_windowed_set_tcbs (ADI_FFT_HANDLE       h,
                                           void                *tcb,
                                           const complex_float  dm input[],
                                           complex_float        dm output[],
                                           const complex_float  pm twiddle[],
                                           int                  twiddle_stride,
                                           const float          dm window[],
                                           int                  n);
```

Description

The `accel_cfft_large_windowed_set_tcbs` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_cfft_large_windowed_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(n, twid_stride)`.

The size of the input array `input`, the `window` array, and the output array `output` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is n. A larger twiddle table may be used, provided that the value of the twiddle table stride argument twiddle_stride is set to the FFT size corresponding to the twiddle table, divided by n.

Twiddle tables accel_twiddles_N, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the libfftacc.dlb library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by accel_cfft_large_windowed_set_tcbs contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both adi_fft_SubmitTxDescriptorChain and adi_fft_SubmitRxDescriptorChain before returning. To subsequently enable the DMA, it is only necessary to call adi_fft_EnableTx and adi_fft_EnableRx. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT  4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
complex_float output[N_FFT];
#pragma align 32
float window[N_FFT];
int   twiddle_stride = 1;
uint32_t tcbmem[ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(N_FFT, 1u)
                                        / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_cfft_large_windowed_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
```

```
   /* Create DMA descriptor chain */
void *tcb = accel_cfft_large_windowed_set_tcbs(h, tcbmem, input, output,
                                               accel_twiddles_4096,
                                               twiddle_stride,
                                               window, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_cfft_large_windowed_pipe, accel_cfft_large_windowed_set_tcbs_strided, accel_fft_set_error_handler, accel_twidfft

accel_cfft_large_windowed_set_tcbs_strided

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) complex radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator, permitting both input and output arrays to be non-contiguous in memory.

Synopsis

```
#include <adi_fft_wrapper.h>
void  *accel_cfft_large_windowed_set_tcbs_strided
                                    (ADI_FFT_HANDLE       h,
                                     void                *tcb,
                                     const complex_float  dm input[],
                                     int                  input_stride,
                                     complex_float        dm output[],
                                     int                  output_stride,
                                     complex_float        dm temp[],
                                     const complex_float  pm twiddle[],
                                     int                  twiddle_stride,
                                     const float          dm window[],
                                     int                  n);
```

Description

The accel_cfft_large_windowed_set_tcbs_strided function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the input array by the window array. The value of n, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_cfft_large_windowed_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(n, twid_stride).

The input array input and the output array output must comprise at least n(strided) data elements. The size of the window array must be at least n. The output and temp arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise they must be aligned to at least an 8 byte boundary. The twiddle array must be aligned to at least a 32 byte boundary. The input array must be aligned to at least an 8 byte boundary. The strides, input_stride and output_stride, specify the stride in data element units; a stride of 1 indicates that the data is contiguous in memory. Best performance is achieved if all contiguous arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is n. A larger twiddle table may be used, provided that the value of the twiddle table stride argument twiddle_stride is set to the FFT size corresponding to the twiddle table, divided by n.

Twiddle tables accel_twiddles_N, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the libfftacc.dlb library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The temp parameter may be used to provide temporary storage which can give improved performance when the value of output_stride is greater than 1. If used, it must point to an array of at least n elements. If not used, the value NULL may be passed in, in which case the output array itself is used for intermediate stages of the FFT computation.

The TX DMA descriptor chain created by accel_cfft_large_windowed_set_tcbs_strided contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both adi_fft_SubmitTxDescriptorChain and adi_fft_SubmitRxDescriptorChain before returning. To subsequently enable the DMA, it is only necessary to call adi_fft_EnableTx and adi_fft_EnableRx. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

complex_float input[2 * N_FFT];
#pragma align 32
complex_float output[256 * N_FFT];
#pragma align 32
float window[N_FFT];
#pragma align 32
complex_float temp[N_FFT];
int   twiddle_stride = 1;
int   input_stride = 2;
int   output_stride = 256;
uint32_t tcbmem[ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(N_FFT, 1u)
                                        / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_cfft_large_windowed_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_cfft_large_windowed_set_tcbs_strided(h, tcbmem,
                                                       input, input_stride,
                                                       output, output_stride,
                                                       temp,
                                                       accel_twiddles_4096,
                                                       twiddle_stride,
                                                       window, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
```

```
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_cfft_large_windowed_pipe, accel_cfft_large_windowed_set_tcbs, accel_fft_set_error_handler, accel_twidfft

accel_cfft_small

Complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_cfft_small (const complex_float  dm input[],
                                  complex_float        dm output[],
                                  float                scale,
                                  int                  n);
```

Description

The `accel_cfft_small` function uses the SHARC+ FFTA accelerator to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048. Due to the overheads associated with setting up the FFTA accelerator for one-time use, it is not recommended to use this function for FFTs of fewer than 1024 points, as it may give poorer performance than computing the FFT on the SHARC+ core.

The size of the input array `input` and the output array `output` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. Best performance is achieved if arrays are aligned to at least a 32 byte boundary.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 1024
```

```
#pragma align 32
complex_float input[N_FFT];
#pragma align 32
complex_float output[N_FFT];
   /* Compute Fast Fourier Transform */
complex_float *result = accel_cfft_small(input, output, 1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_cfft, accel_cfft_large, accel_cfft_small_pipe, accel_fft_set_error_handler

accel_cfft_small_mag_sq

Complex radix-2 Fast Fourier Transform producing magnitude-squared output, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

float  *accel_cfft_small_mag_sq (const complex_float  dm input[],
                                       float          dm output[],
                                       float          scale,
                                       int            n);
```

Description

The `accel_cfft_small_mag_sq` function uses the SHARC+ FFTA accelerator to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048. Due to the overheads associated with setting up the FFTA accelerator for one-time use, it is not recommended to use this function for FFTs of fewer than 1024 points, as it may give poorer performance than computing the FFT on the SHARC+ core.

The size of the input array `input` and the output array `output` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. Best performance is achieved if arrays are aligned to at least a 32 byte boundary.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 1024

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
float         output[N_FFT];
   /* Compute Fast Fourier Transform */
float *result = accel_cfft_small_mag_sq(input, output, 1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_cfft_small, accel_cfft_small_mag_sq_pipe, accel_cfft_small_pipe, accel_fft_set_error_handler

accel_cfft_small_mag_sq_pipe

Set-up function for complex radix-2 Fast Fourier Transform producing magnitude-squared output, using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_cfft_small_mag_sq_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                             float                 scale,
                                             int                   n);
```

Description

The `accel_cfft_small_mag_sq_pipe` function sets up the SHARC+ FFTA accelerator to perform pipelined transformations of the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the input data transfers must be `n` complex data points, while the output data transfers must be `n` real data points. While the results of the FFT may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  2048

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_cfft_small_mag_sq_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
   /* DMA transfers to and from accelerator here */
```

See Also

accel_cfft_small_mag_sq, accel_fft_set_error_handler

accel_cfft_small_pipe

Set-up function for complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>
ADI_FFT_HANDLE accel_cfft_small_pipe (ADI_FFT_DEVICE_MEMORY  *h,
                                      float                  scale,
                                      int                    n);
```

Description

The `accel_cfft_small_pipe` function sets up the SHARC+ FFTA accelerator to perform pipelined transformations of the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the input and output data transfers must be `n` complex data points. While the results of the FFT may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  2048

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_cfft_small_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
   /* DMA transfers to and from accelerator here */
```

See Also

accel_cfft_small, accel_fft_set_error_handler

accel_cfft_small_windowed

Complex radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_cfft_small_windowed (const complex_float  dm input[],
                                           complex_float        dm output[],
                                           const float          dm window[],
                                           float                scale,
                                           int                  n);
```

Description

The `accel_cfft_small_windowed` function uses the SHARC+ FFTA accelerator to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The results are scaled by the value of the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048. Due to the overheads associated with setting up the FFTA accelerator for one-time use, it is not recommended to use this function for FFTs of fewer than 1024 points, as it may give poorer performance than computing the FFT on the SHARC+ core.

The size of the input array `input`, the output array `output` and the window array `window` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. Best performance is achieved if arrays are aligned to at least a 32 byte boundary.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 1024

#pragma align 32
```

```
complex_float input[N_FFT];
#pragma align 32
complex_float output[N_FFT];
#pragma align 32
float         window[N_FFT];
   /* Compute Fast Fourier Transform */
complex_float *result = accel_cfft_small_windowed(input, output,
                                                  window, 1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_cfft_small, accel_cfft_small_pipe, accel_cfft_small_windowed_pipe, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_cfft_small_windowed_mag_sq

Complex radix-2 Fast Fourier Transform with windowing producing magnitude-squared output, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>
complex_float  *accel_cfft_small_windowed_mag_sq
                                 (const complex_float  dm input[],
                                        float          dm output[],
                                  const float          dm window[],
                                        float          scale,
                                        int            n);
```

Description

The `accel_cfft_small_windowed_mag_sq` function uses the SHARC+ FFTA accelerator to transform the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The results are scaled by the value of the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048. Due to the overheads associated with setting up the FFTA accelerator for one-time use, it is not recommended to use this function for FFTs of fewer than 1024 points, as it may give poorer performance than computing the FFT on the SHARC+ core.

The size of the input array `input`, the output array `output` and the window array `window` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. Best performance is achieved if arrays are aligned to at least a 32 byte boundary.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  1024

#pragma align 32
complex_float  input[N_FFT];
#pragma align 32
float   output[N_FFT];
#pragma align 32
float          window[N_FFT];
   /* Compute Fast Fourier Transform */
complex_float *result = accel_cfft_small_windowed_mag_sq(
                                          input, output, window, 1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_cfft_small, accel_cfft_small_windowed_mag_sq_pipe, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_cfft_small_windowed_mag_sq_pipe

Set-up function for complex radix-2 Fast Fourier Transform with windowing producing magnitude-squared output, using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>
ADI_FFT_HANDLE accel_cfft_small_windowed_mag_sq_pipe
                                        (ADI_FFT_DEVICE_MEMORY *h,
                                         const float           dm window[],
                                         float                 scale,
                                         int                   n);
```

Description

The `accel_cfft_small_windowed_mag_sq_pipe` function sets up the SHARC+ FFTA accelerator to perform pipelined transformations of the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. A window function is applied to the input data prior to the FFT transformation by multiplying the input data by the `window` array. The results are scaled by the value of the `scale` parameter. The same window array and scale value is used for all subsequent input data; it is not necessary to send the window array via DMA to the accelerator after calling this function. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the input data transfer must be `n` complex data points, while the size of the output data transfer must be `n` real data points. While the results of the FFT may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  2048

#pragma align 32
float          window[N_FFT];

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
```

```
h = accel_cfft_small_windowed_mag_sq_pipe(&mem, window, 1.0, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_cfft_small_windowed_pipe, accel_cfft_small_windowed_mag_sq, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_cfft_small_windowed_pipe

Set-up function for complex radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_cfft_small_windowed_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                                const float           dm window[],
                                                float                 scale,
                                                int                   n);
```

Description

The `accel_cfft_small_windowed_pipe` function sets up the SHARC+ FFTA accelerator to perform pipelined transformations of the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the input data by the `window` array. The results are scaled by the value of the `scale` parameter. The same window array and scale value is used for all subsequent input data; it is not necessary to send the window array via DMA to the accelerator after calling this function. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the input and output data transfers must be `n` complex data points. While the results of the FFT may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  2048
#pragma align 32

float window[N_FFT];

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_cfft_small_windowed_pipe(&mem, window, 1.0, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_cfft_small_pipe, accel_cfft_small_windowed, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_fft_fir_multi_channel_set_tcbs

Set up DMA descriptor chain to perform DMA transfers for a multi-channel block FIR filter using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_fft_fir_multi_channel_set_tcbs
                               (ADI_FFT_HANDLE      h,
                                void                *tcb,
```

```
                           const float         dm input[],
                           float               dm output[],
                           const complex_float dm coeff[],
                           int                 block_size,
                           int                 ntaps,
                           int                 nchans);
```

Description

The `accel_fft_fir_multi_channel_set_tcbs` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to perform a block FIR filter. The value of `block_size`, which represents the number of points processed per call to the `adi_fft_TriggerTx` function, must be a power of 2 between 32 and 1024. The number of taps, `ntaps`, must be greater than and a multiple of the block size. The FFTA accelerator should be set up first by calling the accel_fft_fir_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_FIR_MULTI_CHANNEL_TCB_CHAIN_SIZE(block_size, ntaps, nchans)`.

The size of the input array `input` must be `block_size` data points, while the output array `output` must be at least `(2 * ntaps)`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The `coeff` array consists of `(2 * ntaps * nchans)` complex FIR filter coefficients that have been transformed to the frequency domain using a sequence of FFT transformations. The coeffients are ordered such that the coefficients for the first channel appear before the coefficients for the second channel, and so on. Ideally, the coefficients should be transformed in advance and linked statically into your program.

Code that shows how to use the single-channel version of this API can be found in the FFT_FIRNonBlocking example in the Board Support Package. To adapt this code for multi-channel use, use a nested loop to send input data to the FFTA, where the outer loop runs over blocks and the inner loop over channels.

The TX DMA descriptor chain created by `accel_fft_fir_multi_channel_set_tcbs` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform FIR filters in non-blocking mode, see Performing a FIR Filter using the FFTA accelerator.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>

#define BLOCK_SIZE  32
#define N_TAPS      512
#define N_CHANS     2
#pragma align 32
float  input[BLOCK_SIZE];
#pragma align 32
float  output[2 * N_TAPS];
#pragma align 32
complex_float  coeff[2 * N_TAPS * N_CHANS];
uint32_t       tcbmem[ACCEL_FFT_FIR_MULTI_CHANNEL_TCB_CHAIN_SIZE
                              (BLOCK_SIZE, N_TAPS, N_CHANS)
                               / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_fft_fir_pipe(&mem, BLOCK_SIZE, N_TAPS);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_fft_fir_multi_channel_set_tcbs
                                  (h, tcbmem, input,
                                   output, coeff,
                                   BLOCK_SIZE, N_TAPS, N_CHANS);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking FIR filter */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_fft_fir_pipe, accel_fft_fir_set_tcbs, accel_fft_set_error_handler

accel_fft_fir_pipe

Set-up function for block FIR filter using the SHARC+ FFTA accelerator in non-blocking mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_fft_fir_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                   int                   block_size,
                                   int                   ntaps);
```

Description

The `accel_fft_fir_pipe` function sets up the SHARC+ FFTA accelerator to perform block FIR filter operations in non-blocking mode. The value of `block_size`, which represents the number of points processed at once by the FIR filter, must be a power of 2 between 32 and 1024. The value of `ntaps`, the number of taps in the filter, must be greater than and a multiple of the block size.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, a DMA descriptor chain must be created to handle the DMA transfers to and from the accelerator. This is done by calling the accel_fft_fir_set_tcbs function. The requirements on data buffers, such as any alignment constraints, are specified in the documentation of that function. For more information on how to perform large FFTs in non-blocking mode, see Using the SHARC+ FFTA Accelerator, and in particular Performing a FIR Filter using the FFTA accelerator.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define BLOCK_SIZE 32
#define N_TAPS  512

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
          /* Set up accelerator for Fast Fourier Transform */
h = accel_fft_fir_pipe(&mem, BLOCK_SIZE, N_TAPS);
if (!h)
```

```
   return 1; /* error! */
         /* Set up DMA descriptor chain here */
```

See Also

accel_fft_fir_set_tcbs, accel_fft_set_error_handler

accel_fft_fir_set_tcbs

Set up DMA descriptor chain to perform DMA transfers for a block FIR filter using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_fft_fir_set_tcbs (ADI_FFT_HANDLE      h,
                               void                *tcb,
                               const float         dm input[],
                               float               dm output[],
                               const complex_float dm coeff[],
                               int                 block_size,
                               int                 ntaps);
```

Description

The `accel_fft_fir_set_tcbs` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to perform a block FIR filter. The value of `block_size`, which represents the number of points processed per call to the `adi_fft_TriggerTx` function, must be a power of 2 between 32 and 1024. The number of taps, `ntaps`, must be greater than and a multiple of the block size. The FFTA accelerator should be set up first by calling the accel_fft_fir_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_FIR_TCB_CHAIN_SIZE(block_size, ntaps)`.

The size of the input array `input` must be `block_size` data points, while the output array `output` must be at least `(2 * ntaps)`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The `coeff` array consists of `(2 * ntaps)` complex FIR filter coefficients that have been transformed to the frequency domain using a sequence of FFT transformations. Code that shows how to do this can be found in the FFT_FIRNonBlocking example in the Board Support Package. Ideally, the coefficients should be transformed in advance and linked statically into your program.

The TX DMA descriptor chain created by `accel_fft_fir_set_tcbs` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform FIR filters in non-blocking mode, see Performing a FIR Filter using the FFTA accelerator.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>

#define BLOCK_SIZE  32
#define N_TAPS      512
#pragma align 32
float  input[BLOCK_SIZE];
#pragma align 32
float  output[2 * N_TAPS];
#pragma align 32

complex_float  coeff[2 * N_TAPS];
uint32_t       tcbmem[ACCEL_FFT_FIR_TCB_CHAIN_SIZE(BLOCK_SIZE, N_TAPS)
                                                  / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_fft_fir_pipe(&mem, BLOCK_SIZE, N_TAPS);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_fft_fir_set_tcbs (h, tcbmem, input,
                                    output, coeff,
                                    BLOCK_SIZE, N_TAPS);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking FIR filter */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
```

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_fft_fir_pipe, accel_fft_fir_multi_channel_set_tcbs, accel_fft_set_error_handler

accel_fft_set_error_handler

Set error handling routine for the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

void accel_fft_set_error_handler (AccelFFTErrorHandler f);
```

Description

The `accel_fft_set_error_handler` function may be used to specify how to handle any errors that occur during use of the SHARC+ FFTA accelerator. It may be useful in order to help to debug any problems that occur when using the `accel_*` family of functions. The parameter passed to `accel_fft_set_error_handler` represents the function that is called if an error occurs.

By default, the error handler is set to `accel_fft_error_handler_null`. This function is used for a reduced code footprint, and does not give any diagnostics regarding the cause of the error.

An alternative error handler, `accel_fft_error_handler_print`, is provided in the FFTA accelerator library. This error handler uses `printf` to output as much information as possible regarding the cause of the error to `stdout`.

If you wish to use your own error handler function, it must be of the following form:

```
void my_fft_error_handler (ADI_FFT_HANDLE     h,
                           ADI_FFT_ERROR_KIND aek,
                           void *             error_handle,
                           unsigned int       error_code) {
    /* error handling code here */
}
```

Note that the interpretation of the parameters is not documented, as these parameters are designed for use by the `accel_fft_error_handler_print` function only.

Error Conditions

None.

Example

```
#include <adi_fft_wrapper.h>

   /* Set error handler */
accel_fft_set_error_handler(accel_fft_error_handler_print);
```

See Also

No related functions

accel_ifft

Inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_ifft (const complex_float  dm input[],
                            complex_float        dm output[],
                            const complex_float  pm twiddle[],
                            int                  twiddle_stride,
                            float                scale,
                            int                  n);
```

Description

The `accel_ifft` function uses the SHARC+ FFTA accelerator to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 4194304. If you know that `n` is less than or equal to 2048, it is better to use the accel_ifft_small function; if `n` is known to be greater than 2048 it is better to use the accel_ifft_large function. Due to the overheads associated with setting up the FFTA accelerator for one-time use, it is not recommended to use this function for FFTs of fewer than 1024 points, as it may give poorer performance than computing the FFT on the SHARC+ core.

The size of the input array `input` and the output array `output` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The other arrays must all be aligned to at least an 8 byte boundary. Best performance is achieved if they are aligned to at least a 32 byte boundary.

The twiddle table is only used for FFTs of 4096 points or more. The minimum size of the twiddle table is n/2 if generated by the twidfft function. A larger twiddle table may be used, provided that the value of the twiddle table stride argument twiddle_stride is set to the FFT size corresponding to the twiddle table, divided by n.

Twiddle tables accel_twiddles_N, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the libfftacc.dlb library. It is most efficient to use these twiddle tables which are optimized for use by the FFTA. Alternatively, the library function twidfft 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 returns the address of the output array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>

#define N_FFT 4096
#pragma align 32
complex_float input[N_FFT];
#pragma align 32

complex_float output[N_FFT];
int twiddle_stride = 1;
   /* Compute Fast Fourier Transform */
complex_float *result = accel_ifft (input, output,
                                    accel_twiddles_4096, twiddle_stride,
                                    1.0/N_FFT, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_ifft_large, accel_ifft_small, accel_ifft_small_pipe, accel_fft_set_error_handler, twidfft

accel_ifftN

Inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>
```

```
complex_float  *accel_ifftN (const complex_float  dm input[],
                             complex_float        dm output[]);
```

Description

The `accel_ifftN` function, where N is one of 1024, 2048, 4096, 8192, 16384, 32768 or 65536, uses the SHARC+ FFTA accelerator to transform 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` and the output array `output` must be at least N. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. Best performance is achieved if arrays are aligned to at least a 32 byte boundary.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#pragma align 32
complex_float input[1024];
#pragma align 32

complex_float output[1024];
   /* Compute Fast Fourier Transform */
complex_float *result = accel_ifft1024(input, output);
if (!result)
   return 1; /* error! */
```

See Also

accel_ifft, accel_ifft_large, accel_ifft_small_pipe, accel_fft_set_error_handler

accel_ifft_large

Inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>
```

```
complex_float  *accel_ifft_large (const complex_float  dm input[],
                                  complex_float        dm output[],
                                  const complex_float  pm twiddle[],
                                  int                  twiddle_stride,
                                  float                scale,
                                  int                  n);
```

Description

The `accel_ifft_large` function uses the SHARC+ FFTA accelerator to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The size of the input array `input` and the output array `output` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA function.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
```

```
complex_float  output[N_FFT];
int twiddle_stride = 1;
   /* Compute Fast Fourier Transform */
complex_float *result = accel_ifft_large(input, output,
                                         accel_twiddles_4096, twiddle_stride,
                                         1.0/N_FFT, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_ifft, accel_ifft_small, accel_ifft_small_pipe, accel_fft_set_error_handler, accel_twidfft

accel_ifft_large_pipe

Set-up function for large (`n` >= 4096) inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator in non-blocking mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_ifft_large_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                      float                  scale,
                                      int                    n);
```

Description

The `accel_ifft_large_pipe` function sets up the SHARC+ FFTA accelerator to perform non-blocking transformations of the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, a DMA descriptor chain must be created to handle the DMA transfers to and from the accelerator. This is done by calling the accel_ifft_large_set_tcbs function. The requirements on data buffers, such as any alignment constraints, are specified in the documentation of that function. For more information on how to perform large FFTs in non-blocking mode, see Using the SHARC+ FFTA Accelerator, and in particular Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  4096

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_ifft_large_pipe(&mem, 1.0, 1.0/N_FFT);
if (!h)
   return 1; /* error! */
 /* Set up DMA descriptor chain here */
```

See Also

accel_ifft_large, accel_ifft_large_set_tcbs, accel_ifft_large_set_tcbs_strided, accel_fft_set_error_handler

accel_ifft_large_set_tcbs

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_ifft_large_set_tcbs (ADI_FFT_HANDLE       h,
                                  void                *tcb,
                                  const complex_float  dm input[],
                                  complex_float        dm output[],
                                  const complex_float  pm twiddle[],
                                  int                  twiddle_stride,
                                  int                  n);
```

Description

The `accel_ifft_large_set_tcbs` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_ifft_large_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(n, twid_stride)`.

The size of the input array `input` and the output array `output` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by `accel_ifft_large_set_tcbs` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
```

```
complex_float output[N_FFT];
int  twiddle_stride = 1;
uint32_t tcbmem[ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(N_FFT, 1u) /
                                                   sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_ifft_large_pipe(&mem, 1.0, 1.0/N_FFT);
if (!h)   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_ifft_large_set_tcbs (h, tcbmem, input,
                                       output, accel_twiddles_4096,
                                       twiddle_stride, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_ifft_large_pipe, accel_ifft_large_set_tcbs_strided, accel_fft_set_error_handler, accel_twidfft

accel_ifft_large_set_tcbs_strided

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator, permitting both input and output arrays to be non-contiguous in memory.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_ifft_large_set_tcbs_strided (ADI_FFT_HANDLE       h,
                                          void                *tcb,
                                          const complex_float  dm input[],
                                          int                  input_stride,
                                          complex_float        dm output[],
                                          int                  output_stride,
                                          complex_float        dm temp[],
                                          const complex_float  pm twiddle[],
```

```
                                   int                 twiddle_stride,
                                   int                 n);
```

Description

The accel_ifft_large_set_tcbs_strided function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). The value of n, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_ifft_large_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(n, twid_stride).

The input array input and the output array output must comprise at least n(strided) data elements. The output and temp arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise they must be aligned to at least an 8 byte boundary. The twiddle array must be aligned to at least a 32 byte boundary. The input array must be aligned to at least an 8 byte boundary. The strides, input_stride and output_stride, specify the stride in data element units; a stride of 1 indicates that the data is contiguous in memory. Best performance is achieved if all contiguous arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is n. A larger twiddle table may be used, provided that the value of the twiddle table stride argument twiddle_stride is set to the FFT size corresponding to the twiddle table, divided by n.

Twiddle tables accel_twiddles_N, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the libfftacc.dlb library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The temp parameter may be used to provide temporary storage which can give improved performance when the value of output_stride is greater than 1. If used, it must point to an array of at least n elements. If not used, the value NULL may be passed in, in which case the output array itself is used for intermediate stages of the FFT computation.

The TX DMA descriptor chain created by accel_ifft_large_set_tcbs_strided contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both adi_fft_SubmitTxDescriptorChain and adi_fft_SubmitRxDescriptorChain before returning. To subsequently enable the DMA, it is only necessary to call adi_fft_EnableTx and adi_fft_EnableRx. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

complex_float input[2 * N_FFT];
#pragma align 32
complex_float output[256 * N_FFT];
#pragma align 32
complex_float temp[N_FFT];
int           twiddle_stride = 1;
int           input_stride = 2;
int           output_stride = 256;
uint32_t      tcbmem[ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(N_FFT, 1u) /
                                                  sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_ifft_large_pipe(&mem, 1.0, 1.0/N_FFT);
if (!h)   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_ifft_large_set_tcbs_strided (h, tcbmem, input, input_stride,
                                               output, output_stride, temp,
                                               accel_twiddles_4096,
                                               twiddle_stride, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
```

```
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_ifft_large_pipe, accel_ifft_large_set_tcbs, accel_fft_set_error_handler, accel_twidfft

accel_ifft_large_windowed

Inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator, with windowing.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_ifft_large_windowed (const complex_float  dm input[],
                                            complex_float        dm output[],
                                            const complex_float  pm twiddle[],
                                            int                  twiddle_stride,
                                            const float          dm window[],
                                            float                scale,
                                            int                  n);
```

Description

The `accel_ifft_large_windowed` function uses the SHARC+ FFTA accelerator to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The results are scaled by the value of the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The size of the input array `input`, the output array `output` and the window array `window` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA function.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
complex_float output[N_FFT];
#pragma align 32
float         window[N_FFT];
int           twiddle_stride = 1;
   /* Compute Fast Fourier Transform */
complex_float *result = accel_ifft_large_windowed
                                        (input, output,
                                         accel_twiddles_4096,
                                         twiddle_stride, window,
                                         1.0/N_FFT, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_ifft_large, accel_ifft_small_windowed, accel_ifft_small_windowed_pipe, accel_fft_set_error_handler, accel_twidfft, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_ifft_large_windowed_pipe

Set-up function for large (`n` >= 4096) inverse complex radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator in non-blocking mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_ifft_large_windowed_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                               float                  scale,
                                               int                    n);
```

Description

The `accel_ifft_large_windowed_pipe` function sets up the SHARC+ FFTA accelerator to perform non-blocking transformations of the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT), with windowing. The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, a DMA descriptor chain must be created to handle the DMA transfers to and from the accelerator. This is done by calling the accel_cfft_large_windowed_set_tcbs function. The requirements on data buffers, such as any alignment constraints, are specified in the documentation of that function. For more information on how to perform large FFTs in non-blocking mode, see Using the SHARC+ FFTA Accelerator, and in particular Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_ifft_large_windowed_pipe(&mem, 1.0, 1.0/N_FFT);
if (!h)
   return 1; /* error! */

 /* Set up DMA descriptor chain here */
```

See Also

accel_ifft_large_windowed, accel_ifft_large_windowed_set_tcbs, accel_ifft_large_windowed_set_tcbs_strided, accel_fft_set_error_handler

accel_ifft_large_windowed_set_tcbs

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) inverse complex radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_ifft_large_windowed_set_tcbs (ADI_FFT_HANDLE       h,
                                           void                *tcb,
                                           const complex_float  dm input[],
                                           complex_float        dm output[],
                                           const complex_float  pm twiddle[],
                                           int                  twiddle_stride,
                                           const float          dm window[],
                                           int                  n);
```

Description

The accel_ifft_large_windowed_set_tcbs function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the input array by the window array. The value of n, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_ifft_large_windowed_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(n, twid_stride).

The size of the input array input, the window array, and the output array output must be at least n. The output array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The twiddle array must be aligned to at least a 32 byte boundary. The input array array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is n. A larger twiddle table may be used, provided that the value of the twiddle table stride argument twiddle_stride is set to the FFT size corresponding to the twiddle table, divided by n.

Twiddle tables accel_twiddles_N, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the libfftacc.dlb library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by `accel_ifft_large_windowed_set_tcbs` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
complex_float output[N_FFT];
#pragma align 32

float         window[N_FFT];
int           twiddle_stride = 1;
uint32_t      tcbmem[ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(N_FFT, 1u) /
                                                             sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_ifft_large_windowed_pipe(&mem, 1.0, 1.0/N_FFT);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_ifft_large_windowed_set_tcbs (h, tcbmem, input,
                                                output, accel_twiddles_4096,
                                                twiddle_stride, window, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
```

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_ifft_large_windowed_pipe, accel_ifft_large_windowed_set_tcbs_strided, accel_fft_set_error_handler, accel_twidfft

accel_ifft_large_windowed_set_tcbs_strided

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) inverse complex radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator, permitting both input and output arrays to be non-contiguous in memory.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_ifft_large_windowed_set_tcbs_strided
                                      (ADI_FFT_HANDLE       h,
                                       void                *tcb,
                                       const complex_float  dm input[],
                                       int                  input_stride,
                                       complex_float        dm output[],
                                       int                  output_stride,
                                       complex_float        dm temp[],
                                       const complex_float  pm twiddle[],
                                       int                  twiddle_stride,
                                       const float          dm window[],
                                       int                  n);
```

Description

The `accel_ifft_large_windowed_set_tcbs_strided` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_ifft_large_windowed_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(n, twid_stride)`.

The input array `input` and the output array `output` must comprise at least `n`(strided) data elements. The size of the `window` array must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise they must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array array must be aligned to at least an 8 byte boundary. The strides, `input_stride` and `output_stride`, specify the stride in data element units; a stride of 1 indicates that the data is contiguous in memory. Best performance is achieved if all contiguous arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The `temp` parameter may be used to provide temporary storage which can give improved performance when the value of `output_stride` is greater than 1. If used, it must point to an array of at least `n` elements. If not used, the value `NULL` may be passed in, in which case the `output` array itself is used for intermediate stages of the FFT computation.

The TX DMA descriptor chain created by `accel_ifft_large_windowed_set_tcbs_strided` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

complex_float input[2 * N_FFT];
#pragma align 32
complex_float output[256 * N_FFT];
#pragma align 32
float         window[N_FFT];
#pragma align 32
complex_float temp[N_FFT];
int           twiddle_stride = 1;
int           input_stride = 2;
int           output_stride = 256;
uint32_t      tcbmem[ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(N_FFT, 1u) /
                                                         sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_ifft_large_windowed_pipe(&mem, 1.0, 1.0/N_FFT);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_ifft_large_windowed_set_tcbs_strided (h, tcbmem,
                                                        input, input_stride,
                                                        output, output_stride,
                                                        temp,
                                                        accel_twiddles_4096,
                                                        twiddle_stride, window,
                                                        N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_ifft_large_windowed_pipe, accel_ifft_large_windowed_set_tcbs, accel_fft_set_error_handler, accel_twidfft

accel_ifft_small

Inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_ifft_small (const complex_float  dm input[],
                                  complex_float        dm output[],
                                  float                scale,
                                  int                  n);
```

Description

The `accel_ifft_small` function uses the SHARC+ FFTA accelerator to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048. Due to the overheads associated with setting up the FFTA accelerator for one-time use, it is not recommended to use this function for FFTs of fewer than 1024 points, as it may give poorer performance than computing the FFT on the SHARC+ core.

The size of the input array `input` and the output array `output` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. Best performance is achieved if arrays are aligned to at least a 32 byte boundary.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  1024
#pragma align 32
complex_float  input[N_FFT];
#pragma align 32

complex_float  output[N_FFT];
   /* Compute Fast Fourier Transform */
complex_float *result = accel_ifft_small(input, output, 1.0 / N_FFT, N_FFT);
```

```
if (!result)
   return 1; /* error! */
```

See Also

accel_ifft, accel_ifft_large, accel_ifft_small_pipe, accel_fft_set_error_handler

accel_ifft_small_pipe

Set-up function for inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_ifft_small_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                      float                 scale,
                                      int                   n);
```

Description

The `accel_ifft_small_pipe` function sets up the SHARC+ FFTA accelerator to perform pipelined transformations of the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the input and output data transfers must be `n` complex data points. While the results of the FFT may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. However you can achieve best performance if you align the buffers to a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  2048

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_ifft_small_pipe(&mem, 1.0/N_FFT, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_ifft_small, accel_fft_set_error_handler

accel_ifft_small_windowed

Inverse complex radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_ifft_small_windowed (const complex_float  dm input[],
                                           complex_float        dm output[],
                                           const float          dm window[],
                                           float                scale,
                                           int                  n);
```

Description

The `accel_ifft_small_windowed` function uses the SHARC+ FFTA accelerator to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The results are scaled by the value of the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048. Due to the overheads associated with setting up the FFTA accelerator for one-time use, it is not recommended to use this function for FFTs of fewer than 1024 points, as it may give poorer performance than computing the FFT on the SHARC+ core.

The size of the input array `input`, the output array `output` and the window array `window` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. Best performance is achieved if arrays are aligned to at least a 32 byte boundary.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  1024
#pragma align 32
complex_float  input[N_FFT];
#pragma align 32
complex_float  output[N_FFT];
#pragma align 32

float          window[N_FFT];
   /* Compute Fast Fourier Transform */
complex_float *result = accel_ifft_small_windowed(input, output, window, 1.0/
N_FFT, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_ifft_small, accel_ifft_small_pipe, accel_ifft_small_windowed_pipe, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_ifft_small_windowed_pipe

Set-up function for inverse complex radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_ifft_small_windowed_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                               const float            dm window[],
```

```
                                                float                   scale,
                                                int                     n);
```

Description

The `accel_ifft_small_windowed_pipe` function sets up the SHARC+ FFTA accelerator to perform pipelined transformations of the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the input data by the `window` array. The results are scaled by the value of the `scale` parameter. The same window array and scale value is used for all subsequent input data; it is not necessary to send the window array via DMA to the accelerator after calling this function. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the input and output data transfers must be `n` complex data points. While the results of the FFT may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  2048
#pragma align 32

float             window[N_FFT];
ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
```

```
   /* Set up accelerator for Fast Fourier Transform */
h = accel_ifft_small_windowed_pipe(&mem, window, 1.0/N_FFT, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_ifft_small_pipe, accel_ifft_small_windowed, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_irfft_large

Real part of inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

float  *accel_irfft_large (const complex_float  dm input[],
                           float                dm output[],
                           complex_float        dm temp[],
                           const complex_float  pm twiddle[],
                           int                  twiddle_stride,
                           float                scale,
                           int                  n);
```

Description

The `accel_irfft_large` function uses the SHARC+ FFTA accelerator to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The real part of the result is written to the output array.

The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The size of the input array `input` and the output array `output` must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory; the `temp` array must in any case be aligned to an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA function.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
float         output[N_FFT];
#pragma align 32
complex_float temp[N_FFT];

int twiddle_stride = 1;
   /* Compute Fast Fourier Transform */
float *result = accel_irfft_large(input, output, temp,
                                  accel_twiddles_4096, twiddle_stride,
                                  1.0/N_FFT, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_irfft_small, accel_irfft_large_pipe, accel_fft_set_error_handler, accel_twidfft

accel_irfft_large_pipe

Set-up function for large (`n` >= 4096) real part of inverse complex radix-2 Fast Fourier Transform using the SHARC + FFTA accelerator in non-blocking mode.

Synopsis

```
#include <adi_fft_wrapper.h>
```

```
ADI_FFT_HANDLE accel_irfft_large_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                       float                 scale,
                                       int                   n);
```

Description

The `accel_irfft_large_pipe` function sets up the SHARC+ FFTA accelerator to perform non-blocking transformations of the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The accelerator produces only the real part of the output array.

The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, a DMA descriptor chain must be created to handle the DMA transfers to and from the accelerator. This is done by calling the accel_irfft_large_set_tcbs function. The requirements on data buffers, such as any alignment constraints, are specified in the documentation of that function. For more information on how to perform large FFTs in non-blocking mode, see Using the SHARC+ FFTA Accelerator, and in particular Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  4096

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_irfft_large_pipe(&mem, 1.0, 1.0/N_FFT);
if (!h)
   return 1; /* error! */
 /* Set up DMA descriptor chain here */
```

See Also

accel_irfft_large, accel_irfft_large_set_tcbs, accel_irfft_large_set_tcbs_strided, accel_fft_set_error_handler

accel_irfft_large_set_tcbs

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) real part of inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_irfft_large_set_tcbs (ADI_FFT_HANDLE       h,
                                   void                *tcb,
                                   const complex_float  dm input[],
                                   float                dm output[],
                                   complex_float        dm temp[],
                                   const complex_float  pm twiddle[],
                                   int                  twiddle_stride,
                                   int                  n);
```

Description

The `accel_irfft_large_set_tcbs` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_irfft_large_pipe function. The accelerator produces only the real part of the output array.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(n, twid_stride)`.

The size of the input array `input`, the output array `output` , and the temporary array `temp` must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory; in any case the `temp` array must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by `accel_irfft_large_set_tcbs` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
float         output[N_FFT];
#pragma align 32
complex_float temp[N_FFT];
int  twiddle_stride = 1;
uint32_t tcbmem[ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(N_FFT, 1u) /
                                                   sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_irfft_large_pipe(&mem, 1.0, 1.0/N_FFT);
if (!h)    return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_irfft_large_set_tcbs (h, tcbmem, input,
                                        output, temp, accel_twiddles_4096,
                                        twiddle_stride, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
```

```
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_irfft_large_pipe, accel_irfft_large_set_tcbs_strided, accel_fft_set_error_handler, accel_twidfft

accel_irfft_large_set_tcbs_strided

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) real part of inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator, permitting both input and output arrays to be non-contiguous in memory.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_irfft_large_set_tcbs_strided (ADI_FFT_HANDLE       h,
                                           void                *tcb,
                                           const complex_float  dm input[],
                                           int                  input_stride,
                                           float                dm output[],
                                           int                  output_stride,
                                           complex_float        dm temp[],
                                           const complex_float  pm twiddle[],
                                           int                  twiddle_stride,
                                           int                  n);
```

Description

The `accel_irfft_large_set_tcbs_strided` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_irfft_large_pipe function. The accelerator produces only the real part of the output array.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(n, twid_stride)`.

The input array `input` and the output array `output` must comprise at least `n`(strided) data elements. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory; in any case the `temp` array must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. The strides, `input_stride` and `output_stride`, specify the stride in data element units; a stride of 1 indicates that the data is contiguous in memory. Best performance is achieved if all contiguous arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The `temp` parameter may be used to provide temporary storage which can give improved performance when the value of `output_stride` is greater than 1. If used, it must point to an array of at least `n` elements. If not used, the value `NULL` may be passed in, in which case the `output` array itself is used for intermediate stages of the FFT computation.

The TX DMA descriptor chain created by `accel_irfft_large_set_tcbs_strided` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

complex_float input[2 * N_FFT];
#pragma align 32
```

```
float          output[256 * N_FFT];
#pragma align 32
complex_float temp[N_FFT];
int            twiddle_stride = 1;
int            input_stride = 2;
int            output_stride = 256;
uint32_t       tcbmem[ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(N_FFT, 1u) /
                                                         sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_irfft_large_pipe(&mem, 1.0, 1.0/N_FFT);
if (!h)   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_irfft_large_set_tcbs_strided (h, tcbmem, input, input_stride,
                                                output, output_stride, temp,
                                                accel_twiddles_4096,
                                                twiddle_stride, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_irfft_large_pipe, accel_irfft_large_set_tcbs, accel_fft_set_error_handler, accel_twidfft

accel_irfft_large_windowed

Real part of inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator, with windowing.

Synopsis

```
#include <adi_fft_wrapper.h>

float  *accel_irfft_large_windowed (const complex_float  dm input[],
                                    float                dm output[],
                                    complex_float        dm temp[],
                                    const complex_float  pm twiddle[],
                                    int                  twiddle_stride,
                                    const float          dm window[],
```

```
                          float                 scale,
                          int                   n);
```

Description

The `accel_irfft_large_windowed` function uses the SHARC+ FFTA accelerator to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The results are scaled by the value of the `scale` parameter. The real part of the result is written to the output array.

The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The size of the input array `input`, the output array `output` and the window array `window` must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory; the `temp` array must in any case be aligned to an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA function.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
float         output[N_FFT];
```

```
#pragma align 32
complex_float temp[N_FFT];
#pragma align 32
float         window[N_FFT];
int           twiddle_stride = 1;
   /* Compute Fast Fourier Transform */
float *result = accel_irfft_large_windowed(input, output, temp,
                                            accel_twiddles_4096,
                                            twiddle_stride, window,
                                            1.0/N_FFT, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_irfft_large, accel_irfft_small_windowed, accel_irfft_small_windowed_pipe, accel_fft_set_error_handler, accel_twidfft, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_irfft_large_windowed_pipe

Set-up function for large (n >= 4096) real part of inverse complex radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator in non-blocking mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_irfft_large_windowed_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                                float                 scale,
                                                int                   n);
```

Description

The `accel_irfft_large_windowed_pipe` function sets up the SHARC+ FFTA accelerator to perform non-blocking transformations of the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT), with windowing. The input values are scaled by the `scale` parameter. The accelerator produces only the real part of the output array.

The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, a DMA descriptor chain must be created to handle the DMA transfers to and from the accelerator. This is done by calling the accel_irfft_large_windowed_set_tcbs function. The requirements on data buffers, such as any alignment constraints, are specified in the documentation of that function. For more information on how to perform large FFTs in non-blocking mode, see Using the SHARC+ FFTA Accelerator, and in particular Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_irfft_large_windowed_pipe(&mem, 1.0, 1.0/N_FFT);
if (!h)
   return 1; /* error! */

 /* Set up DMA descriptor chain here */
```

See Also

accel_irfft_large_windowed, accel_irfft_large_windowed_set_tcbs, accel_irfft_large_windowed_set_tcbs_strided, accel_fft_set_error_handler

accel_irfft_large_windowed_set_tcbs

Set up DMA descriptor chain to perform DMA transfers for large (`n` >= 4096 points) real part of inverse complex radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_irfft_large_windowed_set_tcbs (ADI_FFT_HANDLE       h,
                                            void                *tcb,
                                            const complex_float  dm input[],
                                            float                dm output[],
                                            complex_float        dm temp[],
                                            const complex_float  pm twiddle[],
                                            int                  twiddle_stride,
```

```
                                   const float           dm window[],
                                   int                   n);
```

Description

The `accel_irfft_large_windowed_set_tcbs` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_irfft_large_windowed_pipe function. The accelerator produces only the real part of the output array.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(n, twid_stride).`

The size of the input array `input`, the `window` array, the output array `output` , and the temporary array `temp` must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory; in any case the `temp` array must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by `accel_irfft_large_windowed_set_tcbs` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

#pragma align 32
complex_float input[N_FFT];
#pragma align 32
float         output[N_FFT];
#pragma align 32
complex_float temp[N_FFT];
#pragma align 32
float         window[N_FFT];
int           twiddle_stride = 1;
uint32_t      tcbmem[ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(N_FFT, 1u) /
                                                       sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_irfft_large_windowed_pipe(&mem, 1.0, 1.0/N_FFT);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_irfft_large_windowed_set_tcbs (h, tcbmem, input,
                                                 output, temp,
                                                 accel_twiddles_4096,
                                                 twiddle_stride, window, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_irfft_large_windowed_pipe, accel_irfft_large_windowed_set_tcbs_strided, accel_fft_set_error_handler, accel_twidfft

accel_irfft_large_windowed_set_tcbs_strided

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) real part of inverse complex radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator, permitting both input and output arrays to be non-contiguous in memory.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_irfft_large_windowed_set_tcbs_strided
                                        (ADI_FFT_HANDLE       h,
                                         void                *tcb,
                                         const complex_float  dm input[],
                                         int                  input_stride,
                                         float                dm output[],
                                         int                  output_stride,
                                         complex_float        dm temp[],
                                         const complex_float  pm twiddle[],
                                         int                  twiddle_stride,
                                         const float          dm window[],
                                         int                  n);
```

Description

The `accel_irfft_large_windowed_set_tcbs_strided` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_irfft_large_windowed_pipe function. The accelerator produces only the real part of the output array.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(n, twid_stride)`.

The input array `input` and the output array `output` must comprise at least `n`(strided) data elements. The size of the `window` array must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory; in any case the `temp` array must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array array must be aligned to at least an 8 byte boundary. The strides, `input_stride` and output_stride, specify the stride in data element units; a stride of 1 indicates that the data is contiguous in memory. Best performance is achieved if all contiguous arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is n. A larger twiddle table may be used, provided that the value of the twiddle table stride argument twiddle_stride is set to the FFT size corresponding to the twiddle table, divided by n.

Twiddle tables accel_twiddles_N, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the libfftacc.dlb library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The temp parameter may be used to provide temporary storage which can give improved performance when the value of output_stride is greater than 1. If used, it must point to an array of at least n elements. If not used, the value NULL may be passed in, in which case the output array itself is used for intermediate stages of the FFT computation.

The TX DMA descriptor chain created by accel_irfft_large_windowed_set_tcbs_strided contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both adi_fft_SubmitTxDescriptorChain and adi_fft_SubmitRxDescriptorChain before returning. To subsequently enable the DMA, it is only necessary to call adi_fft_EnableTx and adi_fft_EnableRx. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

complex_float input[2 * N_FFT];
#pragma align 32
float         output[256 * N_FFT];
#pragma align 32
float         window[N_FFT];
#pragma align 32
```

```
complex_float temp[N_FFT];
int           twiddle_stride = 1;
int           input_stride = 2;
int           output_stride = 256;
uint32_t      tcbmem[ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(N_FFT, 1u) /
                                                         sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_irfft_large_windowed_pipe(&mem, 1.0, 1.0/N_FFT);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_irfft_large_windowed_set_tcbs_strided (h, tcbmem,
                                                         input, input_stride,
                                                         output, output_stride,
                                                         temp,
                                                         accel_twiddles_4096,
                                                         twiddle_stride, window,
                                                         N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_irfft_large_windowed_pipe, accel_irfft_large_windowed_set_tcbs, accel_fft_set_error_handler, accel_twidfft

accel_irfft_small

Real part of inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

float  *accel_irfft_small (const complex_float  dm input[],
                           float                dm output[],
                           float                scale,
                           int                  n);
```

Description

The `accel_irfft_small` function uses the SHARC+ FFTA accelerator to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The real part of the result is written to the `output` array.

The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048. Due to the overheads associated with setting up the FFTA accelerator for one-time use, it is not recommended to use this function for FFTs of fewer than 1024 points, as it may give poorer performance than computing the FFT on the SHARC+ core.

The size of the input array `input` and the output array `output` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory. Best performance is achieved if arrays are aligned to at least a 32 byte boundary.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  1024
#pragma align 32
complex_float  input[N_FFT];
#pragma align 32
float  output[N_FFT];

   /* Compute Fast Fourier Transform */
float *result = accel_irfft_small(input, output, 1.0 / N_FFT, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_irfft_large, accel_irfft_small_pipe, accel_fft_set_error_handler

accel_irfft_small_pipe

Set-up function for real part of inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_irfft_small_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                       float                 scale,
                                       int                   n);
```

Description

The `accel_irfft_small_pipe` function sets up the SHARC+ FFTA accelerator to perform pipelined transformations of the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The accelerator produces only the real part of the output array.

The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the input data transfer must be `n` complex data points, while the output data transfer must be n real data points. While the results of the FFT may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. However you can achieve best performance if you align the buffers to a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  2048

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
```

```
   /* Set up accelerator for Fast Fourier Transform */
h = accel_irfft_small_pipe(&mem, 1.0/N_FFT, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_irfft_small, accel_fft_set_error_handler

accel_irfft_small_windowed

Real part of inverse complex radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

float  *accel_irfft_small_windowed (const complex_float  dm input[],
                                    float                dm output[],
                                    const float          dm window[],
                                    float                scale,
                                    int                  n);
```

Description

The `accel_irfft_small_windowed` function uses the SHARC+ FFTA accelerator to transform the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The results are scaled by the value of the `scale` parameter. The real part of the result is written to the output array.

The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048. Due to the overheads associated with setting up the FFTA accelerator for one-time use, it is not recommended to use this function for FFTs of fewer than 1024 points, as it may give poorer performance than computing the FFT on the SHARC+ core.

The size of the input array `input`, the output array `output` and the window array `window` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory. Best performance is achieved if arrays are aligned to at least a 32 byte boundary.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  1024
#pragma align 32
complex_float  input[N_FFT];
#pragma align 32
float          output[N_FFT];
#pragma align 32

float          window[N_FFT];
   /* Compute Fast Fourier Transform */
float *result = accel_irfft_small_windowed(input, output, window, 1.0/N_FFT,
N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_irfft_small, accel_irfft_small_pipe, accel_irfft_small_windowed_pipe, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_irfft_small_windowed_pipe

Set-up function for real part of inverse complex radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_irfft_small_windowed_pipe
                                       (ADI_FFT_DEVICE_MEMORY *h,
                                        const float           dm window[],
                                        float                 scale,
                                        int                   n);
```

Description

The `accel_irfft_small_windowed_pipe` function sets up the SHARC+ FFTA accelerator to perform pipelined transformations of the frequency domain complex input signal sequence to the time domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the input data by the `window` array. The results are scaled by the value of the `scale` parameter. The same window array and scale value is used for all subsequent input data; it is not necessary to send the window array via DMA to the accelerator after calling this function. The accelerator produces only the real part of the output array.

The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the input data transfer must be `n` complex data points, while the output data transfer must be n real data points. . While the results of the FFT may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  2048
#pragma align 32

float          window[N_FFT];
ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_irfft_small_windowed_pipe(&mem, window, 1.0/N_FFT, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_irfft_small_pipe, accel_irfft_small_windowed, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_rfft

Real radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_rfft (const float          dm input[],
                            complex_float        dm output[],
                            const complex_float  pm twiddle[],
                            int                  twiddle_stride,
                            float                scale,
                            int                  n);
```

Description

The `accel_rfft` function uses the SHARC+ FFTA accelerator to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 4194304. If you know that `n` is less than or equal to 2048, it is better to use the accel_rfft_small function; if `n` is known to be greater than 2048 it is better to use the accel_rfft_large function. Due to the overheads associated with setting up the FFTA accelerator for one-time use, it is not recommended to use this function for FFTs of fewer than 1024 points, as it may give poorer performance than computing the FFT on the SHARC+ core.

The size of the input array `input` and the output array `output` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The other arrays must all be aligned to at least an 8 byte boundary. Best performance is achieved if they are aligned to at least a 32 byte boundary.

The twiddle table is only used for FFTs of 4096 points or more. The minimum size of the twiddle table is `n`/2 if generated by the twidfft function. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. It is most efficient to use these twiddle tables which are optimized for use by the FFTA. Alternatively, the library function twidfft 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 returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  4096
#pragma align 32
float          input[N_FFT];
#pragma align 32

complex_float  output[N_FFT];
int            twiddle_stride = 1;
   /* Compute Fast Fourier Transform */
complex_float *result = accel_rfft(input, output, accel_twiddles_4096,
                                   twiddle_stride, 1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_rfft_large, accel_rfft_small, accel_rfft_small_pipe, accel_fft_set_error_handler, twidfft

accel_rfftN

Real radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_rfftN (const float   dm input[],
                             complex_float dm output[]);
```

Description

The accel_rfftN function, where N is one of 1024, 2048, 4096, 8192, 16384, 32768 or 65536, uses the SHARC+ FFTA accelerator to transform 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 output array output must be at least N. The output array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. Best performance is achieved if arrays are aligned to at least a 32 byte boundary.

The function returns the address of the output array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#pragma align 32
float input[1024];
#pragma align 32
complex_float  output[1024];

   /* Compute Fast Fourier Transform */
complex_float *result = accel_rfft1024(input, output);
if (!result)
   return 1; /* error! */
```

See Also

accel_cfftN, accel_ifftN, accel_rfftN, accel_rfft, accel_rfft_large, accel_rfft_small_pipe, accel_fft_set_error_handler

accel_rfft_intleav_small

Set-up function for interleaved real and inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_rfft_intleav_small (ADI_FFT_DEVICE_MEMORY *h,
                                         float                 rscale,
                                         float                 iscale,
                                         int                   n);
```

Description

The `accel_rfft_intleav_small` function sets up the SHARC+ FFTA accelerator to perform pipelined alternate transformations of the time domain real input signal sequence to the frequency domain, and inverse transformations of the frequency domain complex input signal sequence to the time domain, by using the radix-2 Fast Fourier Transform (FFT). The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, h, to the function must point to free memory which is used for storing the state of the FFT device. The second and third parameters, rscale and iscale, specify the scaling factors used for the forward and inverse transformations respectively. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the input data transfers for the forward transformations must be n real data points; all other input and output data transfers must be n complex data points. While the results of the forward and inverse FFTs may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets (one for the forward, and one for the inverse transformation) in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  4096

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_rfft_intleav_small(&mem, 1.0, 1.0/N_FFT, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_cfft_intleav_small, accel_ifft_small, accel_ifft_small_pipe, accel_rfft_intleav_small_mag_sq, accel_rfft_intleav_small_windowed, accel_rfft_intleav_small_windowed_mag_sq, accel_rfft_small, accel_rfft_small_pipe, accel_fft_set_error_handler

accel_rfft_intleav_small_mag_sq

Set-up function for interleaved magnitude-squared real and inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_rfft_intleav_small_mag_sq
                                  (ADI_FFT_DEVICE_MEMORY *h,
                                   float                 rscale,
                                   float                 iscale,
                                   int                   n);
```

Description

The `accel_rfft_intleav_small_mag_sq` function sets up the SHARC+ FFTA accelerator to perform pipelined alternate transformations of the time domain real input signal sequence to the frequency domain, and inverse transformations of the frequency domain complex input signal sequence to the time domain, by using the radix-2 Fast Fourier Transform (FFT). The forward transformation computes the magnitude squared of each element of the output. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The second and third parameters, `rscale` and `iscale`, specify the scaling factors used for the forward and inverse transformations respectively. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the input and output data transfers for the forward transformations must be `n` real data points; the input and output data transfers for the inverse transformations must be `n` complex data points. While the results of the forward and inverse FFTs may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets (one for the forward, and one for the inverse transformation) in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  2048

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_rfft_intleav_small_mag_sq(&mem, 1.0, 1.0 / N_FFT, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_cfft_intleav_small_mag_sq, accel_ifft_small, accel_ifft_small_pipe, accel_rfft_intleav_small, accel_rfft_intleav_small_windowed, accel_rfft_intleav_small_windowed_mag_sq, accel_rfft_small_mag_sq, accel_rfft_small_mag_sq_pipe, accel_fft_set_error_handler

accel_rfft_intleav_small_windowed

Set-up function for interleaved windowed real and inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_rfft_intleav_small_windowed (ADI_FFT_DEVICE_MEMORY *h,
                                                  const float          *window,
                                                  float                 rscale,
                                                  float                 iscale,
                                                  int                   n);
```

Description

The `accel_rfft_intleav_small_windowed` function sets up the SHARC+ FFTA accelerator to perform pipelined alternate transformations of the time domain real input signal sequence to the frequency domain, and inverse transformations of the frequency domain complex input signal sequence to the time domain, by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data to the forward transformation by multiplying the input data by the `window` array. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The third and fourth parameters, `rscale` and `iscale`, specify the scaling factors used for the forward and inverse transformations respectively. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the input data transfers for the forward transformation must be `n` real data points; all other input and output data transfers must be `n` complex data points. While the results of the forward and inverse FFTs may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets (one for the forward, and one for the inverse transformation) in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 2048

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
#pragma align 32
float window[N_FFT];
   /* Set up accelerator for Fast Fourier Transform */
h = accel_rfft_intleav_small_windowed(&mem, window, 1.0, 1.0 /
                                                       N_FFT, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_cfft_intleav_small_windowed, accel_ifft_small_windowed, accel_ifft_small_windowed_pipe, accel_rfft_intleav_small, accel_rfft_intleav_small_mag_sq, accel_rfft_intleav_small_windowed_mag_sq, accel_rfft_small_windowed, accel_rfft_small_windowed_pipe, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_rfft_intleav_small_windowed_mag_sq

Set-up function for interleaved windowed magnitude-squared real and inverse complex radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_rfft_intleav_small_windowed_mag_sq
                                          (ADI_FFT_DEVICE_MEMORY *h,
                                           const float           *window,
                                           float                 rscale,
                                           float                 iscale,
                                           int                   n);
```

Description

The `accel_rfft_intleav_small_windowed_mag_sq` function sets up the SHARC+ FFTA accelerator to perform pipelined alternate transformations of the time domain real input signal sequence to the frequency domain, and inverse transformations of the frequency domain complex input signal sequence to the time domain, by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data to the forward transformation by multiplying the input data by the `window` array, and the magnitude squared is computed for each element of the output. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The third and fourth parameters, `rscale` and `iscale`, specify the scaling factors used for the forward and inverse transformations respectively. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the input and output data transfers for the forward transformation must be `n` real data points; input and output data transfers for the inverse transformation must be `n` complex data points. While the results of the forward and inverse FFTs may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets (one for the forward, and one for the inverse transformation) in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 2048

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
#pragma align 32
float window[N_FFT];
   /* Set up accelerator for Fast Fourier Transform */
h = accel_rfft_intleav_small_windowed_mag_sq(&mem, window, 1.0, 1.0/N_FFT, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_cfft_intleav_small_windowed_mag_sq, accel_ifft_small_windowed, accel_ifft_small_windowed_pipe, accel_rfft_intleav_small, accel_rfft_intleav_small_mag_sq, accel_rfft_intleav_small, accel_rfft_small_windowed, accel_rfft_small_windowed_pipe, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_rfft_large

Real radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>
complex_float  *accel_rfft_large (const float          dm input[],
                                  complex_float        dm output[],
                                  const complex_float  pm twiddle[],
```

```
                              int                     twiddle_stride,
                              float                   scale,
                              int                     n);
```

Description

The accel_rfft_large function uses the SHARC+ FFTA accelerator to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the scale parameter. The value of n, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The size of the input array input and the output array output must be at least n. The output array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The twiddle array must be aligned to at least a 32 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is n. A larger twiddle table may be used, provided that the value of the twiddle table stride argument twiddle_stride is set to the FFT size corresponding to the twiddle table, divided by n.

Twiddle tables accel_twiddles_N, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the libfftacc.dlb library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA function.

The function returns the address of the output array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

#pragma align 32
float         input[N_FFT];
#pragma align 32
complex_float output[N_FFT];
int           twiddle_stride = 1;
   /* Compute Fast Fourier Transform */
complex_float *result = accel_rfft_large(input, output,
                                         accel_twiddles_4096,
```

```
                                           twiddle_stride, 1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_rfft, accel_rfft_small, accel_rfft_small_pipe, accel_fft_set_error_handler, accel_twidfft

accel_rfft_large_mag_sq

Real radix-2 Fast Fourier Transform producing magnitude-squared output, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_rfft_large_mag_sq
                           (const float          dm input[],
                            float                dm output[],
                            complex_float        dm temp[],
                            const complex_float  pm twiddle[],
                            int                  twiddle_stride,
                            float                scale,
                            int                  n);
```

Description

The `accel_rfft_large_mag_sq` function uses the SHARC+ FFTA accelerator to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The size of the input array `input` array, the output array `output` and the `temp` array must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary. The `temp` array is used for temporary storage during the execution of the function.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA function.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

#pragma align 32
float input[N_FFT];
#pragma align 32
float output[N_FFT];
#pragma align 32
complex_float temp[N_FFT];
int twiddle_stride = 1;
   /* Compute Fast Fourier Transform */
complex_float *result = accel_rfft_large_mag_sq
                                  (input, output, temp,
                                   accel_twiddles_4096, twiddle_stride,
                                   1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_rfft_large, accel_rfft_small_mag_sq, accel_rfft_small_mag_sq_pipe, accel_fft_set_error_handler, accel_twidfft

accel_rfft_large_mag_sq_pipe

Set-up function for large (`n` >= 4096) real radix-2 Fast Fourier Transform producing magnitude-squared output, using the SHARC+ FFTA accelerator in non-blocking mode.

Synopsis

```
#include <adi_fft_wrapper.h>
```

```
ADI_FFT_HANDLE accel_rfft_large_mag_sq_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                             float                  scale,
                                             int                    n);
```

Description

The accel_rfft_large_mag_sq_pipe function sets up the SHARC+ FFTA accelerator to perform non-blocking transformations of the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. The input values are scaled by the scale parameter. The value of n, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The first parameter, h, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, a DMA descriptor chain must be created to handle the DMA transfers to and from the accelerator. This is done by calling the accel_rfft_large_mag_sq_set_tcbs function. The requirements on data buffers, such as any alignment constraints, are specified in the documentation of that function. For more information on how to perform large FFTs in non-blocking mode, see Using the SHARC+ FFTA Accelerator, and in particular Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_rfft_large_mag_sq_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */

 /* Set up DMA descriptor chain here */
```

See Also

accel_rfft_large_mag_sq, accel_rfft_large_mag_sq_set_tcbs, accel_rfft_large_mag_sq_set_tcbs_strided, accel_fft_set_error_handler

accel_rfft_large_mag_sq_set_tcbs

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) real radix-2 Fast Fourier Transform producing magnitude-squared output, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_rfft_large_mag_sq_set_tcbs
                                  (ADI_FFT_HANDLE       h,
                                   void                 *tcb,
                                   const float          dm input[],
                                   float                dm output[],
                                   complex_float        dm temp[],
                                   const complex_float  pm twiddle[],
                                   int                  twiddle_stride,
                                   int                  n);
```

Description

The `accel_rfft_large_set_tcbs` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_rfft_large_mag_sq_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_MAG_SQ_TCB_CHAIN_SIZE(n, twid_stride)`.

The size of the input array `input`, the output array `output` and the `temp` array must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise they must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary. The `temp` array is used for temporary storage during computation of the FFT.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by `accel_rft_large_mag_sq_set_tcbs` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

#pragma align 32
float         input[N_FFT];
#pragma align 32
float         output[N_FFT];
#pragma align 32
complex_float temp[N_FFT];
int           twiddle_stride = 1;
uint32_t      tcbmem[ACCEL_FFT_LARGE_MAG_SQ_TCB_CHAIN_SIZE(N_FFT, 1u)
                                  / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_rfft_large_mag_sq_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_rfft_large_mag_sq_set_tcbs
                                    (h, tcbmem, input,
                                     output, temp, accel_twiddles_4096,
```

```
                                          twiddle_stride, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_rfft_large_mag_sq_pipe, accel_rfft_large_mag_sq_set_tcbs_strided, accel_fft_set_error_handler, accel_twidfft

accel_rfft_large_mag_sq_set_tcbs_strided

Set up DMA descriptor chain to perform DMA transfers for large (`n` >= 4096 points) real radix-2 Fast Fourier Transform producing magnitude-squared output, using the SHARC+ FFTA accelerator, permitting both input and output arrays to be non-contiguous in memory.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_rfft_large_mag_sq_set_tcbs_strided
                                  (ADI_FFT_HANDLE       h,
                                   void                 *tcb,
                                   const float          dm input[],
                                   int                  input_stride,
                                   float                dm output[],
                                   int                  output_stride,
                                   complex_float        dm temp[],
                                   const complex_float  pm twiddle[],
                                   int                  twiddle_stride,
                                   int                  n);
```

Description

The `accel_rfft_large_set_tcbs_strided` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_rfft_large_mag_sq_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_MAG_SQ_TCB_CHAIN_SIZE(n, twid_stride)`.

The input array `input` and the output array `output` must comprise at least `n`(strided) data elements. The size of the `temp` array must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise they must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The strides, `input_stride` and `output_stride`, specify the stride in data element units; a stride of 1 indicates that the data is contiguous in memory. Best performance is achieved if all contiguous arrays are aligned to at least a 32 byte boundary. The `temp` array is used for temporary storage during computation of the FFT.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by `accel_rft_large_mag_sq_set_tcbs_strided` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
```

```
#define N_FFT 4096

float         input[2 * N_FFT];
#pragma align 32
float         output[256 * N_FFT];
#pragma align 32
complex_float temp[N_FFT];
int           twiddle_stride = 1;
int           input_stride = 2;
int           output_stride = 256;
uint32_t      tcbmem[ACCEL_FFT_LARGE_MAG_SQ_TCB_CHAIN_SIZE(N_FFT, 1u)
                                                 / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_rfft_large_mag_sq_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_rfft_large_mag_sq_set_tcbs_strided
                                    (h, tcbmem, input, input_stride,
                                     output, output_stride, temp,
                                     accel_twiddles_4096,
                                     twiddle_stride, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_rfft_large_mag_sq_pipe, accel_rfft_large_mag_sq_set_tcbs, accel_fft_set_error_handler, accel_twidfft

accel_rfft_large_pipe

Set-up function for large (n >= 4096) real radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator in non-blocking mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_rfft_large_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                      float                 scale,
                                      int                   n);
```

Description

The `accel_rfft_large_pipe` function sets up the SHARC+ FFTA accelerator to perform non-blocking transformations of the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, a DMA descriptor chain must be created to handle the DMA transfers to and from the accelerator. This is done by calling the accel_rfft_large_set_tcbs function. The requirements on data buffers, such as any alignment constraints, are specified in the documentation of that function. For more information on how to perform large FFTs in non-blocking mode, see Using the SHARC+ FFTA Accelerator, and in particular Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_rfft_large_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */

 /* Set up DMA descriptor chain here */
```

See Also

accel_rfft_large, accel_rfft_large_set_tcbs, accel_cfft_large_set_tcbs_strided, accel_fft_set_error_handler

accel_rfft_large_set_tcbs

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) real radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_rfft_large_set_tcbs (ADI_FFT_HANDLE       h,
                                  void                *tcb,
                                  const float          dm input[],
                                  complex_float        dm output[],
                                  const complex_float  pm twiddle[],
                                  int                  twiddle_stride,
                                  int                  n);
```

Description

The accel_rfft_large_set_tcbs function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). The value of n, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_rfft_large_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(n, twid_stride).

The size of the input array input and the output array output must be at least n. The output array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The twiddle array must be aligned to at least a 32 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is n. A larger twiddle table may be used, provided that the value of the twiddle table stride argument twiddle_stride is set to the FFT size corresponding to the twiddle table, divided by n.

Twiddle tables accel_twiddles_N, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the libfftacc.dlb library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by `accel_rfft_large_set_tcbs` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>

#define N_FFT 4096
#pragma align 32
float input[N_FFT];
#pragma align 32
complex_float output[N_FFT];
int twiddle_stride = 1;
uint32_t tcbmem[ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(N_FFT, 1u)
                             / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_rfft_large_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_rfft_large_set_tcbs (h, tcbmem, input,
                                       output, accel_twiddles_4096,
                                       twiddle_stride, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
```

```
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_rfft_large_pipe, accel_rfft_large_set_tcbs_strided, accel_fft_set_error_handler, accel_twidfft

accel_rfft_large_set_tcbs_strided

Set up DMA descriptor chain to perform DMA transfers for large (`n` >= 4096 points) real radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator, permitting both input and output arrays to be non-contiguous in memory.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_rfft_large_set_tcbs_strided (ADI_FFT_HANDLE       h,
                                          void                *tcb,
                                          const float          dm input[],
                                          int                  input_stride,
                                          complex_float        dm output[],
                                          int                  output_stride,
                                          complex_float        temp[],
                                          const complex_float  pm twiddle[],
                                          int                  twiddle_stride,
                                          int                  n);
```

Description

The `accel_rfft_large_set_tcbs_strided` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_rfft_large_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(n, twid_stride)`.

The input array `input` and the output array `output` must comprise at least `n`(strided) data elements. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise they must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The strides, `input_stride` and `output_stride`, specify the stride in data element units; a stride of 1 indicates that the data is contiguous in memory. Best performance is achieved if all contiguous arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The `temp` parameter may be used to provide temporary storage which can give improved performance when the value of `output_stride` is greater than 1. If used, it must point to an array of at least `n` elements. If not used, the value `NULL` may be passed in, in which case the `output` array itself is used for intermediate stages of the FFT computation.

The TX DMA descriptor chain created by `accel_rfft_large_set_tcbs_strided` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>

#define N_FFT 4096
#pragma align 32
float input[2 * N_FFT];
#pragma align 32
complex_float output[256 * N_FFT];
#pragma align 32
complex_float temp[N_FFT];
```

```
int twiddle_stride = 1;
int input_stride = 2;
int output_stride = 256;
uint32_t tcbmem[ACCEL_FFT_LARGE_TCB_CHAIN_SIZE(N_FFT, 1u)
                             / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_rfft_large_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_rfft_large_set_tcbs_strided (h, tcbmem, input, input_stride,
                                               output, output_stride, temp,
                                               accel_twiddles_4096,
                                               twiddle_stride, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_rfft_large_pipe, accel_rfft_large_set_tcbs, accel_fft_set_error_handler, accel_twidfft

accel_rfft_large_windowed

Real radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator, with windowing.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_rfft_large_windowed (const float          dm input[],
                                           complex_float        dm output[],
                                           const complex_float  pm twiddle[],
                                           int                  twiddle_stride,
                                           const float          dm window[],
                                           float                scale,
                                           int                  n);
```

Description

The accel_rfft_large_windowed function uses the SHARC+ FFTA accelerator to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the input array by the window array. The results are scaled by the value of the scale parameter. The value of n, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The size of the input array input, the output array output and the window array window must be at least n. The output array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The twiddle array must be aligned to at least a 32 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is n. A larger twiddle table may be used, provided that the value of the twiddle table stride argument twiddle_stride is set to the FFT size corresponding to the twiddle table, divided by n.

Twiddle tables accel_twiddles_N, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the libfftacc.dlb library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA function.

The function returns the address of the output array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

#pragma align 32
float input[N_FFT];
#pragma align 32
complex_float output[N_FFT];
#pragma align 32
float window[N_FFT];
int twiddle_stride = 1;
   /* Compute Fast Fourier Transform */
complex_float *result = accel_rfft_large_windowed
                                    (input, output,
```

```
                                    accel_twiddles_4096,
                                    twiddle_stride,
                                    window, 1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_rfft_large, accel_rfft_small_windowed, accel_rfft_small_windowed_pipe, accel_fft_set_error_handler, accel_twidfft, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_rfft_large_windowed_mag_sq

Real radix-2 Fast Fourier Transform producing magnitude-squared output, using the SHARC+ FFTA accelerator, with windowing.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_rfft_large_windowed_mag_sq
                                    (const float          dm input[],
                                     float                dm output[],
                                     complex_float        dm temp[],
                                     const complex_float  pm twiddle[],
                                     int                  twiddle_stride,
                                     const float          dm window[],
                                     float                scale,
                                     int                  n);
```

Description

The `accel_rfft_large_windowed` function uses the SHARC+ FFTA accelerator to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The results are scaled by the value of the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The size of the input array `input`, the output array `output`, the window array `window` and the `temp` array must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary. The `temp` array is used as temporary storage during the execution of the function.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA function.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

#pragma align 32
float input[N_FFT];
#pragma align 32
float output[N_FFT];
#pragma align 32
complex_float temp[N_FFT];
#pragma align 32
float window[N_FFT];
int twiddle_stride = 1;
   /* Compute Fast Fourier Transform */
complex_float *result = accel_rfft_large_windowed_mag_sq
                                         (input, output,
                                          temp, accel_twiddles_4096,
                                          twiddle_stride, window,
                                          1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_rfft_large_mag_sq, accel_rfft_small_windowed_mag_sq, accel_rfft_small_windowed_mag_sq_pipe, accel_fft_set_error_handler, accel_twidfft, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_rfft_large_windowed_mag_sq_pipe

Set-up function for large (n >= 4096) real radix-2 Fast Fourier Transform with windowing and producing magnitude-squared output, using the SHARC+ FFTA accelerator in non-blocking mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_rfft_large_windowed_mag_sq_pipe
                                                    (ADI_FFT_DEVICE_MEMORY *h,
                                                     float                 scale,
                                                     int                   n);
```

Description

The `accel_rfft_large_windowed_mag_sq_pipe` function sets up the SHARC+ FFTA accelerator to perform non-blocking transformations of the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), with windowing, and computes the magnitude-squared of each element of the output. The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, a DMA descriptor chain must be created to handle the DMA transfers to and from the accelerator. This is done by calling the accel_rfft_large_windowed_set_tcbs function. The requirements on data buffers, such as any alignment constraints, are specified in the documentation of that function. For more information on how to perform large FFTs in non-blocking mode, see Using the SHARC+ FFTA Accelerator, and in particular Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
```

```
h = accel_rfft_large_windowed_mag_sq_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */

 /* Set up DMA descriptor chain here */
```

See Also

accel_rfft_large_windowed_mag_sq, accel_rfft_large_windowed_mag_sq_set_tcbs, accel_rfft_large_windowed_mag_sq_set_tcbs_strided, accel_fft_set_error_handler

accel_rfft_large_windowed_mag_sq_set_tcbs

Set up DMA descriptor chain to perform DMA transfers for large (`n` >= 4096 points) real radix-2 Fast Fourier Transform with windowing and producing magnitude-squared output, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_rfft_large_windowed_mag_sq_set_tcbs (
                                     ADI_FFT_HANDLE       h,
                                     void                *tcb,
                                     const float          dm input[],
                                     float                dm output[],
                                     complex_float        dm temp[],
                                     const complex_float  pm twiddle[],
                                     int                  twiddle_stride,
                                     const float          dm window[],
                                     int                  n);
```

Description

The `accel_rfft_large_windowed_mag_sq_set_tcbs` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_rfft_large_windowed_mag_sq_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_WINDOWED_MAG_SQ_TCB_CHAIN_SIZE(n, twid_stride)`.

The size of the input array `input`, the `window` array, the output array `output` and the `temp` array must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise they must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary. The `temp` array is used for temporary storage during computation of the FFT.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by `accel_rfft_large_windowed_mag_sq_set_tcbs` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

#pragma align 32
float input[N_FFT];
#pragma align 32
float output[N_FFT];
#pragma align 32
float window[N_FFT];
#pragma align 32
complex_float temp[N_FFT];
int  twiddle_stride = 1;
```

```
uint32_t  tcbmem[ACCEL_FFT_LARGE_WINDOWED_MAG_SQ_TCB_CHAIN_SIZE(N_FFT, 1u)
                      / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
    /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_rfft_large_windowed_mag_sq_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_rfft_large_windowed_mag_sq_set_tcbs
                                         (h, tcbmem, input,
                                          output, temp, accel_twiddles_4096,
                                          twiddle_stride, window, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_rfft_large_windowed_mag_sq_pipe, accel_rfft_large_windowed_mag_sq_set_tcbs_strided, accel_fft_set_error_handler, accel_twidfft

accel_rfft_large_windowed_mag_sq_set_tcbs_strided

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) real radix-2 Fast Fourier Transform with windowing and producing magnitude-squared output, using the SHARC+ FFTA accelerator, permitting both input and output arrays to be non-contiguous in memory.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_rfft_large_windowed_mag_sq_set_tcbs_strided (
                                      ADI_FFT_HANDLE       h,
                                      void                *tcb,
                                      const float          dm input[],
                                      int                  input_stride,
                                      float                dm output[],
                                      int                  output_stride,
```

```
                          complex_float        dm temp[],
                          const complex_float  pm twiddle[],
                          int                  twiddle_stride,
                          const float          dm window[],
                          int                  n);
```

Description

The `accel_rfft_large_windowed_mag_sq_set_tcbs_strided` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_rfft_large_windowed_mag_sq_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_WINDOWED_MAG_SQ_TCB_CHAIN_SIZE(n, twid_stride)`.

The input array `input` and the output array `output` must comprise at least `n`(strided) data elements. The size of the the `window` array and the `temp` array must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise they must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The `input` array must be aligned to at least an 8 byte boundary. The strides, `input_stride` and `output_stride`, specify the stride in data element units; a stride of 1 indicates that the data is contiguous in memory. Best performance is achieved if all contiguous arrays are aligned to at least a 32 byte boundary. The `temp` array is used for temporary storage during computation of the FFT.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by `accel_rfft_large_windowed_mag_sq_set_tcbs_strided` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>
#define N_FFT 4096

float input[2 * N_FFT];
#pragma align 32
float output[256 * N_FFT];
#pragma align 32
float window[N_FFT];
#pragma align 32
complex_float temp[N_FFT];
int  twiddle_stride = 1;
int  input_stride = 2;
int  output_stride = 256;

uint32_t  tcbmem[ACCEL_FFT_LARGE_WINDOWED_MAG_SQ_TCB_CHAIN_SIZE(N_FFT, 1u)
                      / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
    /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_rfft_large_windowed_mag_sq_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_rfft_large_windowed_mag_sq_set_tcbs_strided
                                         (h, tcbmem, input, input_stride,
                                          output, output_stride, temp,
                                          accel_twiddles_4096,
                                          twiddle_stride, window, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

```
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_rfft_large_windowed_mag_sq_pipe, accel_rfft_large_windowed_mag_sq_set_tcbs, accel_fft_set_error_handler, accel_twidfft

accel_rfft_large_windowed_pipe

Set-up function for large (n >= 4096) real radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator in non-blocking mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_rfft_large_windowed_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                               float                  scale,
                                               int                    n);
```

Description

The `accel_rfft_large_windowed_pipe` function sets up the SHARC+ FFTA accelerator to perform non-blocking transformations of the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), with windowing. The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, a DMA descriptor chain must be created to handle the DMA transfers to and from the accelerator. This is done by calling the accel_rfft_large_windowed_set_tcbs function. The requirements on data buffers, such as any alignment constraints, are specified in the documentation of that function. For more information on how to perform large FFTs in non-blocking mode, see Using the SHARC+ FFTA Accelerator, and in particular Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 4096

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_rfft_large_windowed_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */

 /* Set up DMA descriptor chain here */
```

See Also

accel_rfft_large_windowed, accel_rfft_large_windowed_set_tcbs, accel_rfft_large_windowed_set_tcbs_strided, accel_fft_set_error_handler

accel_rfft_large_windowed_set_tcbs

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) real radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_rfft_large_windowed_set_tcbs (ADI_FFT_HANDLE       h,
                                           void                *tcb,
                                           const float          dm input[],
                                           complex_float        dm output[],
                                           const complex_float  pm twiddle[],
                                           int                  twiddle_stride,
                                           const float          dm window[],
                                           int                  n);
```

Description

The accel_rfft_large_windowed_set_tcbs function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the input array by the window array. The value of n, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_rfft_large_windowed_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(n, twid_stride)`.

The size of the input array `input`, the `window` array, and the output array `output` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. Best performance is achieved if all arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The TX DMA descriptor chain created by `accel_rfft_large_windowed_set_tcbs` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>

#define N_FFT 4096
```

```
#pragma align 32
float input[N_FFT];
#pragma align 32
complex_float output[N_FFT];
#pragma align 32
float window[N_FFT];
int twiddle_stride = 1;
uint32_t tcbmem[ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(N_FFT, 1u)
                                / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_rfft_large_windowed_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_rfft_large_windowed_set_tcbs(h, tcbmem, input,
                                               output, accel_twiddles_4096,
                                               twiddle_stride, window, N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_rfft_large_windowed_pipe, accel_rfft_large_windowed_set_tcbs_strided, accel_fft_set_error_handler, accel_twidfft

accel_rfft_large_windowed_set_tcbs_strided

Set up DMA descriptor chain to perform DMA transfers for large (n >= 4096 points) real radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator, permitting both input and output arrays to be non-contiguous in memory.

Synopsis

```
#include <adi_fft_wrapper.h>

void  *accel_rfft_large_windowed_set_tcbs_strided
                                    (ADI_FFT_HANDLE      h,
```

```
                                            void                 *tcb,
                                            const float           dm input[],
                                            int                   input_stride,
                                            complex_float         dm output[],
                                            int                   output_stride,
                                            complex_float         dm temp[],
                                            const complex_float   pm twiddle[],
                                            int                   twiddle_stride,
                                            const float           dm window[],
                                            int                   n);
```

Description

The `accel_rfft_large_windowed_set_tcbs_strided` function sets up a DMA descriptor chain in order to use the SHARC+ FFTA accelerator in non-blocking mode to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 4096 and 4194304. The FFTA accelerator should be set up first by calling the accel_rfft_large_windowed_pipe function.

Space for the DMA descriptor chain is allocated by the caller and a pointer to the space is passed to the function as its second parameter. The amount of space required, in bytes, is determined by the macro `ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(n, twid_stride).`

The input array `input` and the output array `output` must comprise at least `n`(strided) data elements. The size of the `window` array must be at least `n`. The `output` and `temp` arrays must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise they must be aligned to at least an 8 byte boundary. The `twiddle` array must be aligned to at least a 32 byte boundary. The strides, `input_stride` and `output_stride`, specify the stride in data element units; a stride of 1 indicates that the data is contiguous in memory. Best performance is achieved if all contiguous arrays are aligned to at least a 32 byte boundary.

The minimum size of the twiddle table is `n`. A larger twiddle table may be used, provided that the value of the twiddle table stride argument `twiddle_stride` is set to the FFT size corresponding to the twiddle table, divided by `n`.

Twiddle tables `accel_twiddles_N`, where N is one of 4096, 8192, 16384, 32768 or 65536, may be found in the `libfftacc.dlb` library. These twiddle tables are pre-generated and optimized for use by the FFTA. Alternatively, the library function accel_twidfft 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, with the twiddles ordered according to their use in the FFTA algorithm.

The `temp` parameter may be used to provide temporary storage which can give improved performance when the value of `output_stride` is greater than 1. If used, it must point to an array of at least `n` elements. If not used, the value `NULL` may be passed in, in which case the `output` array itself is used for intermediate stages of the FFT computation.

The TX DMA descriptor chain created by `accel_rfft_large_windowed_set_tcbs_strided` contains synchronization points where the next DMA transfer waits for the RX DMA to raise a trigger. Consequently you must initialize the Trigger Routing Unit (TRU) and then register the RX Trigger Master as follows:

```
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
```

The function calls both `adi_fft_SubmitTxDescriptorChain` and `adi_fft_SubmitRxDescriptorChain` before returning. To subsequently enable the DMA, it is only necessary to call `adi_fft_EnableTx` and `adi_fft_EnableRx`. Both the TX and RX DMA channels will then wait for a trigger before proceeding.

The function returns the address of the DMA descriptor chain. For more information on how to use the FFTA to perform large FFTs in non-blocking mode, see Using Large FFTs in Continuous Mode.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#include <services/tru/adi_tru.h>

#define N_FFT  4096
float input[2 * N_FFT];
#pragma align 32
complex_float output[256 * N_FFT];
#pragma align 32
float window[N_FFT];
#pragma align 32
complex_float temp[N_FFT];
int twiddle_stride = 1;
int input_stride = 2;
int output_stride = 256;
uint32_t tcbmem[ACCEL_FFT_LARGE_WINDOWED_TCB_CHAIN_SIZE(N_FFT, 1u)
                                              / sizeof(uint32_t)];
ADI_FFT_DEVICE_MEMORY mem;
   /* Set up FFTA for non-blocking large FFT */
ADI_FFT_HANDLE h = accel_rfft_large_windowed_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */
   /* Create DMA descriptor chain */
void *tcb = accel_rfft_large_windowed_set_tcbs_strided(h, tcbmem,
                                                       input, input_stride,
                                                       output, output_stride,
                                                       temp,
                                                       accel_twiddles_4096,
                                                       twiddle_stride, window,
```

```
N_FFT);
if (!tcb)
   return 1; /* error! */
   /* Set up TRU for non-blocking large FFT */
if (adi_tru_Init(/*reset*/true) != ADI_TRU_SUCCESS)
   return 1; /* error! */
adi_fft_RegisterTxTriggerMaster(h, TRGM_FFTA0_RXDMA);
adi_fft_RegisterRxTriggerMaster(h, TRGM_SYS_SOFT1_MST);
   /* Enable DMA transfers */
adi_fft_EnableTx(h, true);
adi_fft_EnableRx(h, true);
   /* Trigger DMA channels to start transfers */
adi_fft_TriggerTx(h);
adi_fft_TriggerRx(h);
```

See Also

accel_rfft_large_windowed_pipe, accel_rfft_large_windowed_set_tcbs, accel_fft_set_error_handler, accel_twidfft

accel_rfft_small

Real radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_rfft_small (const float          dm input[],
                                  complex_float        dm output[],
                                  float                scale,
                                  int                  n);
```

Description

The `accel_rfft_small` function uses the SHARC+ FFTA accelerator to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048. Due to the overheads associated with setting up the FFTA accelerator for one-time use, it is not recommended to use this function for FFTs of fewer than 1024 points, as it may give poorer performance than computing the FFT on the SHARC+ core.

The size of the input array `input` and the output array `output` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. Best performance is achieved if arrays are aligned to at least a 32 byte boundary.

The function returns the address of the `output` array.

NOTE: The FFTA accelerator must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>

#define N_FFT 1024
#pragma align 32
float   input[N_FFT];
#pragma align 32
complex_float output[N_FFT];
   /* Compute Fast Fourier Transform */
complex_float *result = accel_rfft_small(input, output, 1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_rfft, accel_rfft_large, accel_rfft_small_pipe, accel_fft_set_error_handler

accel_rfft_small_mag_sq

Real radix-2 Fast Fourier Transform producing magnitude-squared output, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

float  *accel_rfft_small_mag_sq (const float  dm input[],
                                 float        dm output[],
                                 float        scale,
                                 int          n);
```

Description

The `accel_rfft_small_mag_sq` function uses the SHARC+ FFTA accelerator to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048. Due to the overheads associated with setting up the FFTA accelerator for one-time use, it is not recommended to use this function for FFTs of fewer than 1024 points, as it may give poorer performance than computing the FFT on the SHARC+ core.

The size of the input array `input` and the output array `output` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. Best performance is achieved if arrays are aligned to at least a 32 byte boundary.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>

#define N_FFT 1024
#pragma align 32
float input[N_FFT];
#pragma align 32
float output[N_FFT];
   /* Compute Fast Fourier Transform */
float *result = accel_rfft_small_mag_sq(input, output, 1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_rfft_small, accel_rfft_small_mag_sq_pipe, accel_rfft_small_pipe, accel_fft_set_error_handler

accel_rfft_small_mag_sq_pipe

Set-up function for real radix-2 Fast Fourier Transform producing magnitude-squared output, using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_rfft_small_mag_sq_pipe
                              (ADI_FFT_DEVICE_MEMORY *h,
                               float                 scale,
                               int                   n);
```

Description

The `accel_rfft_small_mag_sq_pipe` function sets up the SHARC+ FFTA accelerator to perform pipelined transformations of the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the input and output data transfers must be `n` real data points. While the results of the FFT may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 2048

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_rfft_small_mag_sq_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_rfft_small_mag_sq, accel_fft_set_error_handler

accel_rfft_small_pipe

Set-up function for real radix-2 Fast Fourier Transform using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_rfft_small_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                      float                 scale,
                                      int                   n);
```

Description

The `accel_rfft_small_pipe` function sets up the SHARC+ FFTA accelerator to perform pipelined transformations of the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). The input values are scaled by the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the input data transfers must be `n` real data point, whilethe output data transfers must be `n` complex data points. While the results of the FFT may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 2048

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_rfft_small_pipe(&mem, 1.0, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_rfft_small, accel_fft_set_error_handler

accel_rfft_small_windowed

Real radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_rfft_small_windowed
                          (const float    dm input[],
                           complex_float  dm output[],
                           const float    dm window[],
                           float          scale,
                           int            n);
```

Description

The `accel_rfft_small_windowed` function uses the SHARC+ FFTA accelerator to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The results are scaled by the value of the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048. Due to the overheads associated with setting up the FFTA accelerator for one-time use, it is not recommended to use this function for FFTs of fewer than 1024 points, as it may give poorer performance than computing the FFT on the SHARC+ core.

The size of the input array `input`, the output array `output` and the window array `window` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. Best performance is achieved if arrays are aligned to at least a 32 byte boundary.

The function returns the address of the output array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 1024

#pragma align 32
float input[N_FFT];
#pragma align 32
complex_float output[N_FFT];
#pragma align 32
float window[N_FFT];
   /* Compute Fast Fourier Transform */
complex_float *result = accel_rfft_small_windowed
                       (input, output, window, 1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_rfft_small, accel_rfft_small_pipe, accel_rfft_small_windowed_pipe, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_rfft_small_windowed_mag_sq

Real radix-2 Fast Fourier Transform with windowing producing magnitude-squared output, using the SHARC+ FFTA accelerator.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float  *accel_rfft_small_windowed_mag_sq (const float    dm input[],
                                                  float          dm output[],
                                                  const float    dm window[],
                                                  float          scale,
                                                  int            n);
```

Description

The `accel_rfft_small_windowed_mag_sq` function uses the SHARC+ FFTA accelerator to transform the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. A window function is applied to the input data prior to the FFT transformation by multiplying the `input` array by the `window` array. The results are scaled by the value of the `scale` parameter. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048. Due to the overheads associated with setting up the FFTA accelerator for one-time use, it is not recommended to use this function for FFTs of fewer than 1024 points, as it may give poorer performance than computing the FFT on the SHARC+ core.

The size of the input array `input`, the output array `output` and the window array `window` must be at least `n`. The `output` array must be aligned to the cache line size (64 bytes on the SHARC core, 32 bytes on ARM) if placed in cacheable memory, otherwise it must be aligned to at least an 8 byte boundary. Best performance is achieved if arrays are aligned to at least a 32 byte boundary.

The function returns the address of the `output` array.

NOTE: The FFTA must not be in use when this function is called.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  1024

#pragma align 32
float input[N_FFT];
#pragma align 32
float output[N_FFT];
#pragma align 32
float window[N_FFT];
   /* Compute Fast Fourier Transform */
float *result = accel_rfft_small_windowed_mag_sq
                         (input, output, window, 1.0, N_FFT);
if (!result)
   return 1; /* error! */
```

See Also

accel_rfft_small_windowed, accel_rfft_small_windowed_mag_sq_pipe, accel_rfft_small_windowed_pipe, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_rfft_small_windowed_mag_sq_pipe

Set-up function for real radix-2 Fast Fourier Transform with windowing producing magnitude-squared output, using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_rfft_small_windowed_mag_sq_pipe
                                  (ADI_FFT_DEVICE_MEMORY *h,
                                   const float           dm window[],
                                   float                 scale,
                                   int                   n);
```

Description

The `accel_rfft_small_windowed_mag_sq_pipe` function sets up the SHARC+ FFTA accelerator to perform pipelined transformations of the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT), and computes the magnitude-squared of each element of the output. A window function is applied to the input data prior to the FFT transformation by multiplying the input data by the `window` array. The results are scaled by the value of the `scale` parameter. The same window array and scale value is used for all subsequent input data; it is not necessary to send the window array via DMA to the accelerator after calling this function. The value of `n`, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, `h`, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the both the input and output data transfers must be `n` real data points. While the results of the FFT may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns `NULL`. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT  2048
#pragma align 32

float window[N_FFT];
ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_rfft_small_windowed_mag_sq_pipe(&mem, window, 1.0, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_rfft_small_windowed_mag_sq, accel_rfft_small_windowed_pipe, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_rfft_small_windowed_pipe

Set-up function for real radix-2 Fast Fourier Transform with windowing, using the SHARC+ FFTA accelerator in pipelined mode.

Synopsis

```
#include <adi_fft_wrapper.h>

ADI_FFT_HANDLE accel_rfft_small_windowed_pipe (ADI_FFT_DEVICE_MEMORY *h,
                                               const float           dm window[],
                                               float                 scale,
                                               int                   n);
```

Description

The `accel_rfft_small_windowed_pipe` function sets up the SHARC+ FFTA accelerator to perform pipelined transformations of the time domain real input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT). A window function is applied to the input data prior to the FFT transformation by multiplying the input data by the `window` array. The results are scaled by the value of the `scale` parameter. The same window array and scale value is used for all subsequent input data; it is not necessary to send the window array via DMA to the accelerator after calling this function. The value of n, which represents the number of points in the FFT, must be a power of 2 between 64 and 2048.

The first parameter, h, to the function must point to free memory which is used for storing the state of the FFT device. The function returns a handle to the FFT accelerator device.

NOTE: The FFTA must not be in use when this function is called.

Data Requirements

After the FFTA is initialized using this function, data must be sent to and received from the accelerator using DMA. This can be done in two ways, and is described in Continuous and Pipelined Operation. The size of the input data transfers must be n real data points, while the output data transfers must be n complex data points. While the results of the FFT may be received after transmitting the corresponding input data, maximum throughput is obtained by submitting two data sets in advance. This pipelining allows concurrent use of the input and output DMA channels and FFTA compute hardware. For more information, see Using the SHARC+ FFTA Accelerator.

After calling the function, the DMA transfer size for both the TX and RX channels is set to the default value of 8 bytes, which requires that the input and output data buffers be aligned to at least an 8 byte boundary. In addition, if the output data buffer is located in cacheable memory, you should align this buffer to the cache line size (64 bytes on SHARC core, 32 bytes on ARM). For best performance align both buffers to at least a 32 byte boundary, and set the DMA transfer size to 32 bytes.

Error Conditions

If an error is encountered, the function returns NULL. The accel_fft_set_error_handler function can be used to register an error handler to indicate the cause of the error.

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 2048

#pragma align 32
float window[N_FFT];

ADI_FFT_DEVICE_MEMORY mem;
ADI_FFT_HANDLE h;
   /* Set up accelerator for Fast Fourier Transform */
h = accel_rfft_small_windowed_pipe(&mem, window, 1.0, N_FFT);
if (!h)
   return 1; /* error! */

   /* DMA transfers to and from accelerator here */
```

See Also

accel_rfft_small_pipe, accel_rfft_small_windowed, accel_fft_set_error_handler, gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

accel_twidfft

Generate FFT twiddle factors for use by the FFTA.

Synopsis

```
#include <adi_fft_wrapper.h>

complex_float pm *accel_twidfft (complex_float pm twiddle_tab[],
                                 int           fftsize);
```

Description

The `accel_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 FFTA functions to calculate a Fast Fourier Transform. The table generated by this function may be used by any of the "large" FFTA functions, `accel_{cir}fft_large_*`, but is not suitable for FFT functions that do not run on the FFTA. 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 `accel_{cir}fft_large*` has 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 first parameter passed to the `accel_twidfft` function must be a pointer to a free array of `fftsize` complex data points which is aligned to at least a 32-byte boundary.

The function returns a pointer to `twiddle_tab`, or `NULL` if an error occurred.

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-1}

These twiddle coefficients are then reordered in the twiddle table according to how they are used by the FFTA functions.

Error Conditions

None

Example

```
#include <adi_fft_wrapper.h>
#define N_FFT 131072

complex_float in[N_FFT];
complex_float out[N_FFT];
complex_float pm twid_tab[N_FFT];

accel_twidfft (twid_tab, N_FFT);
accel_cfft_large (in, out, twid_tab, 1, 1.0, N_FFT);
```

See Also

twidfftf

a_compress

A-law compression

Synopsis (Scalar-Valued Version)

```
#include <comm.h>

int a_compress (int x);
```

Synopsis (Vector-Valued Version)

```
#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.

NOTE: The vector-valued version of `a_compress` uses serial port 0 to perform the compressing 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

None

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];
a_compress (data, compressed, NSAMPLES);
```

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)

```
#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.

> **NOTE:** The vector version of the `a_expand` function uses serial port 0 to perform the expanding 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

None

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]; a_expand (compressed_data, expanded_data, NSAMPLES);

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

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

arg

Get phase of a complex number

Synopsis

```
#include <cmatrix.h>

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

Description

The `arg function` 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

cartesian

autocoh

Auto-coherence

Synopsis

```
#include <stats.h>

float *autocohf (float output[],const float  input[],
                 intsamples, int lags);
double *autocoh (double output[], const double input[],
                 int samples, int lags);
long double *autocohd (long double output[],
                       const long double input[],
                       int samples, int lags);
```

Description

The `autocoh` functions compute the auto-coherence of the signal contained in the array `input` of length `samples`. The auto-coherence of an input signal is its auto-correlation minus the product of the partial means of the input signal.

The auto-coherence between the input signal and itself is returned in the array `output` of length `lags`. The functions return a pointer to the output array.

Error Conditions

The `autocoh` functions will return without modifying the output array if either the number of samples is less than or equal to 1, or if the number of lags is less than 1, or if the number of lags is not less than the number of samples.

Algorithm

The auto-coherence functions are based on the following algorithm.

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

where:

`n` = `samples`

`k` = `0` to `lags-1`

`a` = `input`

Example

```
#include <stats.h>
#define SAMPLES 1024
#define LAGS      16

float x[SAMPLES];float response[LAGS];
autocohf (response, x, 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.

NOTE: The `autocorrf` function (and `autocorr`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode 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

None

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)

```
#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`

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. The definition of the state is:

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

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.

CAUTION: 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 or ADSP-21369 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-2136x 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

$$D_m = A_2 \bullet D_{m-2} + A_1 \bullet D_{m-1} + x_m$$

$$Y_m = B_2 \bullet D_{m-2} + B_1 \bullet D_{m-1} + B_0 \bullet D_m$$

where:

`m = {0,1,2,...,samples-1}`

The algorithm used is adapted from *Digital Signal Processing*, Oppenheim and Schafer, New Jersey, Prentice Hall, 1975. For more information, see the ***Biquad Sections*** figure.

Error Conditions

None

Example

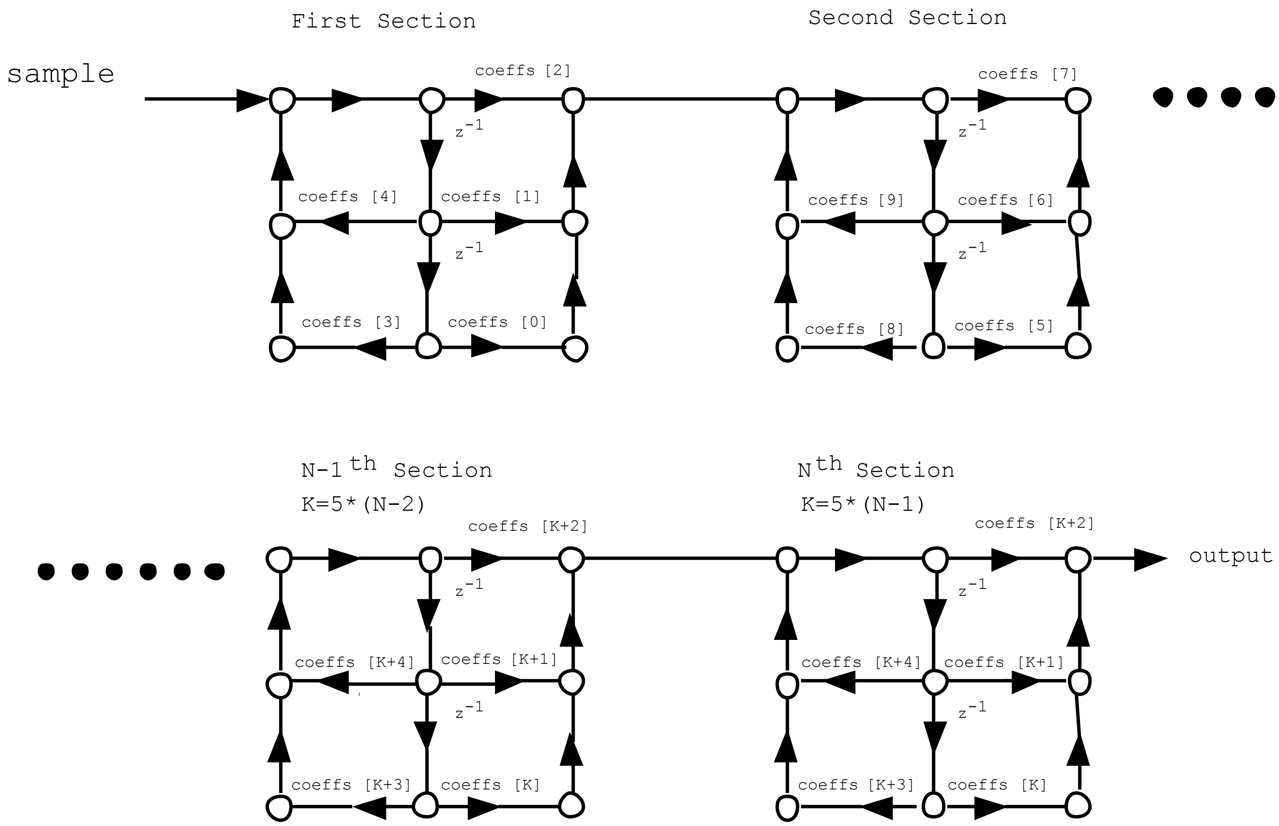
Scalar-Valued

```
#include <filters.h>
#define NSECTIONS  4
#define NSTATE    (2*NSECTIONS)

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)

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 */

biquad (input, output, coeffs, state, NSAMPLES,
        NSECTIONS);
```

Note that N = the number of biquad sections.



Biquad Sections

See Also

biquad_trans, fir, iir

biquad_trans

Biquad filter section

Synopsis

```
#include <filter.h>

float *biquad_trans (       float   dm data[],
                      const float   pm coeffs[],
                            float   dm state[],
                            int        samples,
                            int        sections);
```

Description

The biquad_trans function implements a cascaded biquad filter defined by the coefficients and the number of sections that are supplied in the call to the function. The biquad uses the transposed direct form II algorithm.

The biquad_trans function generates the filtered response of the input data data and stores the result in the same vector data. The number of input samples and the length of this input/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_trans function assumes that the value of A0 is 1.0, and A1 and A2 should be scaled accordingly. These coefficients are passed to the biquad_trans function 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 biquad_trans function, the coefficients for each pair of sections must be interleaved and stored in the order:

B0, B1, B2, A1, A2

If you have an odd number of sections, the last section's coefficients should be stored sequentially in the obove order. For example, for a five-section filter (labeled below as sections s0 to s4), the coefficients would be stored in memory as follows:

B0_s0, B0_s1, B1_s0, B1_s1, B2_s0, B2_s1, A1_s0, A1_s1, A2_s0, A2_s1, B0_s2, B0_s3, B1_s2, B1_s3, B2_s2, B2_s3, A1_s2, A1_s3, A2_s2, A2_s3, B0_s4, B1_s4, B2_s4, A1_s4, A2_s4

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. The definition of the state is:

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

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 biquad_trans function returns a pointer to the input/output vector.

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:

m = {`0,1,2,...,samples-1`}

The algorithm used is adapted from *Digital Signal Processing*, Oppenheim and Schafer, New Jersey, Prentice Hall, 1975. For more information, see the ***Biquad Sections*** figure.

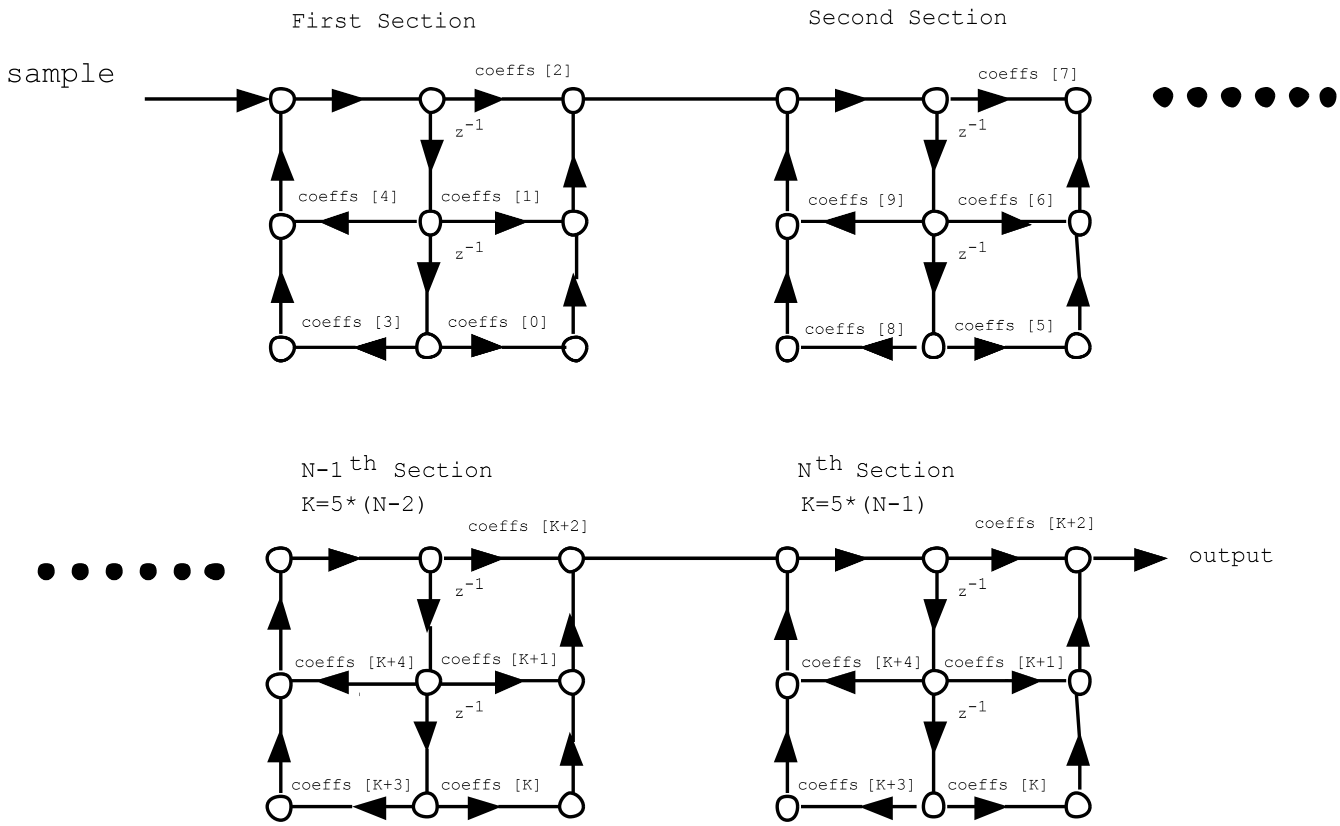
Error Conditions

None

Example

```
#include <filter.h>
#define NSECTIONS  4
#define NSAMPLES  64
#define NSTATE    (2*NSECTIONS)

float data[NSAMPLES];

float state[NSTATE];
float pm coeffs[5*NSECTIONS];
int i;

for (i = 0; i < NSTATE; i++)
   state[i] = 0;     /* initialize state array    */
biquad_trans (data, coeffs, state, NSAMPLES, NSECTIONS);
```

Note that N = the number of biquad sections.



Biquad Sections

See Also

biquad, fir, iir

cabs

Complex absolute value

Synopsis

```
#include <cmatrix.h>

float cabsf (complex float z);
double cabs (complex double z);
long double cabsl (complex long double z);

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.

$$y = \sqrt{\left((Re(z))^2 + (Im(z))^2\right)}$$

The functions which use the `complex` keyword as only available when building in C99 mode. The `cabs` and `cabsf` fucntions use the same name for both C99 native complex types and `complex_float` / `complex_double` types. A type-generic macro is used to select the correct function based on the operand type. This mechanism however does not permit the address of the function to be taken. If you wish to take the address of the function, please define either `ADI_COMPLEX_STRUCT_FORM` (for the structure type support) or `ADI_COMPLEX_C99` (for C99 native complex) which will avoid use of the type-generic macro.

Error Conditions

None

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

No additional references

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

None

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

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

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

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 cexpl (complex long double z);

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));`

The functions which use the `complex` keyword as only available when building in C99 mode. The `cexp` and `cexpf` fucntions use the same name for both C99 native complex types and `complex_float` / `complex_double` types. A type-generic macro is used to select the correct function based on the operand type. This mechanism however does not permit the address of the function to be taken. If you wish to take the address of the function, please define either `ADI_COMPLEX_STRUCT_FORM` (for the structure type support) or `ADI_COMPLEX_C99` (for C99 native complex) which will avoid use of the type-generic macro.

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

No additional references

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 to the FFT size of the table generated divided by the size of the FFT being performed. If the size of the twiddle table is `x`, then `twiddle_stride` must be set to (2*`x`)/`n`.

The library function twidfft 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.

NOTE: The library also contains the cfftf function, 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.

NOTE: The `cfft` function uses SIMD by default. Refer to Implications of Using SIMD Mode 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

None

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

accel_cfft, accel_ifft, accel_rfft, cfftf, cfftN, fft_magnitude, ifft, rfft, twidfft

cfftf

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 be 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 twidfftffunction 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.

NOTE: The `cfftf` function has been implemented to make highly efficient use of the processor's SIMD capabilities. 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.
- Due to the alignment restrictions of the input arrays (as documented above), it is unlikely that the function will generate the correct results if the input arrays are allocated on the stack.

For more information, refer to Implications of Using SIMD Mode.

Error Conditions

None

Example

```
#include <filter.h>
#define FFT_SIZE 1024
#pragma align 1024
static float dm input_r[FFT_SIZE];
#pragma align 1024

static float pm input_i[FFT_SIZE];
float dm temp_r[FFT_SIZE];
float pm temp_i[FFT_SIZE+1];
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, fftf_magnitude, ifftf, rfftf_2, twidfftf

cfft_mag

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.

NOTE: 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

None

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, fft_magnitude , fftf_magnitude, rfft_mag

NOTE: By default, this function uses SIMD. Refer to Implications of Using SIMD Mode 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.

NOTE: 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.

Error Conditions

None

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, fft_magnitude, ifftN, rfftN

cfftN

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 SHARC processors supported by CCES. 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.

NOTE: 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

None

Example

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

complex_float input[N], output[N];

cfft2048 (input, output);
```

See Also

accel_cfftN, accel_ifftN, accel_rfftN, cfft, cfftf, fft_magnitude, ifftN, rfftN

NOTE: By default these functions use SIMD. For more information, refer to Implications of Using SIMD Mode.

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

None

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

NOTE: The `cmatmadf` function (and `cmatmadd`, if `doubles` are the same size as `floats`) uses SIMD. Refer to Implications of Using SIMD Mode 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

None

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

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

None

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

NOTE: The `cmatmsubf` function (and `cmatmsub`, if `doubles` are the same size as `floats`) uses SIMD. Refer to Implications of Using SIMD Mode 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

None

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

NOTE: The `cmatsaddf` function (and `cmatsadd`, if `doubles` are the same size as `floats`) uses SIMD. Refer to Implications of Using SIMD Mode 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 by scalar multiplication is defined by the following algorithm:

$$Re(c_{i,j}) = Re(a_{i,j}) \times Re(scalar) - Im(a_{i,j}) \times Im(scalar)$$

$$Im(c_{i,j}) = Re(a_{i,j}) \times Im(scalar) + Im(a_{i,j}) \times Re(scalar)$$

where:

i = {0,1,2,..., rows-1}

j = {0,1,2,..., cols-1}

Error Conditions

None

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

None

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

NOTE: The `cmatssubf` function (and `cmatssub`, if `doubles` are the same size as `floats`) uses SIMD; refer to Implications of Using SIMD Mode 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

None

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 conjl (complex long double a);

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.

The functions which use the complex keyword as only available when building in C99 mode. The conj and conjf fucntions use the same name for both C99 native complex types and complex_float / complex_double types. A type-generic macro is used to select the correct function based on the operand type. This mechanism however does not permit the address of the function to be taken. If you wish to take the address of the function, please define either ADI_COMPLEX_STRUCT_FORM (for the structure type support) or ADI_COMPLEX_C99 (for C99 native complex) which will avoid use of the type-generic macro.

Error Conditions

None

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 functions

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

None

Example

```
#include <filter.h>

float input[81];
float response[31];
float output[81 + 31 -1];

convolve(input,81,response,31,output);
```

See Also

crosscorr

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

None

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 functions

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

No related functions

crosscoh

Cross-coherence

Synopsis

```
#include <stats.h>

float *crosscohf (float output[],
                  const float x_input[],
                  const float y_input[],
                  int samples,
                  int lags);

double *crosscoh (double output[],
                  const double x_input[],
                  const double y_input[],
                  int samples,
                  int lags);

long double *crosscohd (long double output[],
                        const long double x_input[],
                        const long double y_input[],
                        int samples,
                        int lags);
```

Description

The `crosscoh` functions perform a cross-coherence between the two signals contained in `x_input` and `y_input`, both of length `samples`. The cross-coherence is the sum of the scalar products of the input signals in which the signals are displaced in time with respect to one another (i.e. the cross-correlation between the input signals), minus the product of the partial mean of `x_input` and the partial mean of `y_input`.

The cross-coherence between the two input signals is returned in the array `output` of length `lags`. The functions return a pointer to the output array.

Error Conditions

The `crosscoh` functions will return without modifying the output array if either the number of samples is less than or equal to 1, or if the number of lags is less than 1, or if the number of lags is not less than the number of samples.

Algorithm

The cross-coherence functions are based on the following algorithm.

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

where:

n = samples

k = 0 to lags-1

a = x_input

b = y_input

Example

```
#include <stats.h>
#define SAMPLES 1024
#define LAGS    16

float x[SAMPLES];
float y[SAMPLES];
float response[LAGS];

crosscohf (response, x, y, SAMPLES, LAGS);
```

See Also

autocoh, autocorr, crosscorr

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

None

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

NOTE: The `crosscorrf` function (and `crosscorr`, if `doubles` are the same size as `floats`) uses SIMD; refer to Implications of Using SIMD Mode 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

None

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:

$$Re(c_i) = \sum_{l=0}^{n-1} (Re(a_i) \bullet (Re(b_i)) - Im(a_i) \bullet Im(b_i))$$

$$Im(c_i) = \sum_{l=0}^{n-1} (Re(a_i) \bullet (Im(b_i)) + Im(a_i) \bullet Re(b_i))$$

where:

i = {0,1,2,...,samples-1}

Error Conditions

None

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 <cmatrix.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

None

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

NOTE: The cvecsaddf function (and cvecsadd, if doubles are the same size as floats) uses SIMD by default. Refer to Implications of Using SIMD Mode 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

None

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

None

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

NOTE: The `cvecssubf` function (and `cvecssub`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode 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

None

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

NOTE: The `cvecvaddf` function (and `cvecvadd`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode 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:

$\mathrm{Re}(c_i) = \mathrm{Re}(a_i)*\mathrm{Re}(b_i) - \mathrm{Im}(a_i)*\mathrm{Im}(b_i)$

$\mathrm{Im}(c_i) = \mathrm{Re}(a_i)*\mathrm{Im}(b_i) + \mathrm{Im}(a_i)*\mathrm{Re}(b_i)$

i = {0,1,2,...,samples-1}

Error Conditions

None

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

NOTE: Restrictions apply to this function if the data is placed in external memory. Refer to Using Data in External Memory for more information. Also the input and output arrays for cvecvmltf, and cvecvmlt when doubles are floats, must be long-word aligned as use is made of long-word (LW) memory accesses. The best way of ensuring correct alignment of these arrays is to define them statically or globally as automatic arrays in interrupt handlers may not always be aligned.

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

None

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

NOTE: The `cvecvsubf` function (and `cvecvsub`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode for more information.

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

None

Example

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

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

See Also

No related functions

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

None

Example

```
#include <math.h>

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

See Also

No related functions

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)`.

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.

> **NOTE:** 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).
>
> The `fft_magnitude` function uses SIMD by default. Refer to Implications of Using SIMD Mode for more information.

Error Conditions

None

Algorithm

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);

rfft64 (rfft_input, rfftN_output);
fft_magnitude (rfftN_output, rNspectrum, N_FFT, 1);

/* 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, cfft_mag, fftf_magnitude, rfft, rfft_mag, rfftN

ffff_magnitude

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 will contain a signal that is symmetrical about its midpoint and so the function will only use the first `((fftsize / 2) + 1)` input samples to compute the power spectrum. The size of the generated power spectrum will be `((fftsize / 2) + 1)`.

The `fftf_magnitude` function returns a pointer to the output.

Algorithm

For mode 0 (`cfftf` generated input):

$$magnitude(z) = \frac{\sqrt{Re(a_z)^2 + Im(a_z)^2}}{fftsize}$$

For mode 2 (`rfftf_2` generated input):

$$magnitude(z) = 2 \times \frac{\sqrt{Re(a_z)^2 + Im(a_z)^2}}{fftsize}$$

Error Conditions

None

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, rfftf_2

NOTE: By default, this function uses SIMD. Refer to Implications of Using SIMD Mode 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)

```
#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 (pass band, 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.

CAUTION: 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 or ADSP-21369 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-2136x 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

None

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       */

fir (input, output, coeffs, state, SAMPLES, TAPS);
```

See Also

biquad, fir_decima, fir_interp, firf, iir

NOTE: By default, the vector version of the `fir` function uses SIMD. Refer to Implications of Using SIMD Mode for more information.

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.

CAUTION: 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 or ADSP-21369 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-2136x 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

None

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;

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

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.

CAUTION: 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 or ADSP-21369 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-2136x 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

None

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

firf

Fast Finite Impulse Response (FIR) filter

Synopsis

```
#include <filter.h>

void *firf (const float input[],
            float output[],
            const float pm coefficients[],
            float state[],
            int samples,
            int taps);
```

Description

The `firf` function implements an accelerated finite impulse response (FIR) filter. The function generates the filtered response of the input data `input` and stores the result in the output vector `output`. The number of input samples and the length of the output vector are specified by the parameter `samples`. The number of samples must be even and at least 4. The function will ignore the last sample if the number of samples is odd.

The number of coefficients is specified by the parameters `taps`. The number of coefficients must be even and at least 8. If the number of filter coefficients is odd, then an application could round the number of coefficients up to the next even number and set the extra coefficient to 0. The filter coefficients must be stored in reverse order in the array `coefficients`. Thus `coefficients[0]` contains the last filter coefficient and `coefficients[taps-1]` contains the first coefficient. The array should be located in a different memory section than the `state` array (see below) 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 length of the `state` array is the number of taps + 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.

CAUTION: The library function uses the architecture's dual-data move instructions to provide simultaneous access to the filter coefficients (in PM data memory) and the delay line. When running on a ADSP-21367, ADSP-21368, or ADSP-21369 processor, the delay line and filter coefficients must not both be allocated in external memory otherwise the function may generate an incorrect set of results. This may happen because the hardware does not support a dual-data move instruction that generates two accesses to external memory. Therefore ensure that either the filter coefficients or the delay line (or, optimally, both) are allocated in internal memory when running on one of the ADSP-2136x processors specified above.

To provide optimal performance, the function uses the architecture's SIMD mode and also makes use of certain user-reservable registers. It is therefore important to note the following constraints concerning the use of the function:

- Refer to Implications of Using SIMD Mode and ***A Brief Introduction to SIMD Mode*** in the *C/C++ Compiler Manual for SHARC Processors* for further information concerning the use of SIMD. A non-SIMD version of this function is not provided.
- Under the C/C++ run-time model, certain registers are defined as *reservable registers* (refer to ***Reservable Registers*** in the *C/C++ Compiler Manual for SHARC Processors*). Normally the run-time library will avoid using these registers.

  However, the `firf` function will make use of the following registers, preserving their contents on entry to the function and restoring them on exit:

  ```
  B1, I1, L1
  ```

  Therefore applications that use the compiler's `-reserve` switch to reserve the above registers should not use the `firf` function.

The constraints imposed by the `firf` function may not meet the requirements of an application; in cases such as these, applications can instead use the fir function which has the same functionality as the firf function but none of its restrictions.

The function returns a pointer to the output array.

Algorithm

The algorithm is based on:

```
output[i] = sum (h[k] * x[i-j])
```

where:

```
x = input
h = array of coefficients
i = { 0, 1, ..., samples-1 }
j = { 0, 1, ..., taps-1 }
k = { taps-1, taps-2, ..., 0 }
```

Error Conditions

None

Example

```
#include <filter.h>

#define TAPS     64
#define SAMPLES 512
```

```
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       */
for (;;)
{
   /* acquire a new set of input data */
   /* compute filtered response for current set of data */
   firf (input, output, coeffs, state, SAMPLES, TAPS);
   /* post-process filtered response */
}
```

See Also

fir

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 `fmaxf` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

None

Example

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

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

See Also

fmin

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

None

Example

```
#include <math.h>

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

See Also

fmax

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) 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

None

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

None

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

None

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

None

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

None

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

None

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[], float beta, 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`}

a = `(N - 1) / 2`

I_0(b) represents the zeroth-order modified Bessel function of the first kind

Domain

a > 0; N > 0; b > 0.0

Error Conditions

None

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

None

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 in gen_bartlett) 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

None

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 the `gen_hanning` window (described in gen_hanning).

Error Conditions

None

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

None

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

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 `twidfft` 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` 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.

NOTE: The library also contains the `ifftf` function (see ifftf), 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

None

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

accel_cfft, accel_ifft, accel_rfft, cfft, ifftf, ifftN, rfft, twidfft

NOTE: The ifft function uses SIMD by default. Refer to Implications of Using SIMD Mode for more information.

ifftf

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 be 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 twidfftffunction 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.

NOTE: The `ifftf` function has been implemented to make highly efficient use of the processor's SIMD capabilities. 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.
- Due to the alignment restrictions of the input arrays (as documented above), it is unlikely that the function will generate the correct results if the input arrays are allocated on the stack.

For more information, refer to Implications of Using SIMD Mode.

Error Conditions

None

Example

```
#include <filter.h>
#define FFT_SIZE 1024

#pragma align 1024
static float dm input_r[FFT_SIZE];
#pragma align 1024
static float pm input_i[FFT_SIZE];
float dm temp_r[FFT_SIZE];
float pm temp_i[FFT_SIZE+1];
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, ifft, ifftN, rfftf_2, twidfftf

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.

NOTE: 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 ifftN instead.

Error Conditions

None

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, rfftN

ifftN

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 SHARC processors. 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.

NOTE: 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

None

Example

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

complex_float input[N], output[N];
ifft2048 (input, output);
```

See Also

accel_cfftN, accel_ifftN, accel_rfftN, cfftN, ifft, ifftf, rfftN

NOTE: By default, these functions use SIMD. Refer to Implications of Using SIMD Mode 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)

```
#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.

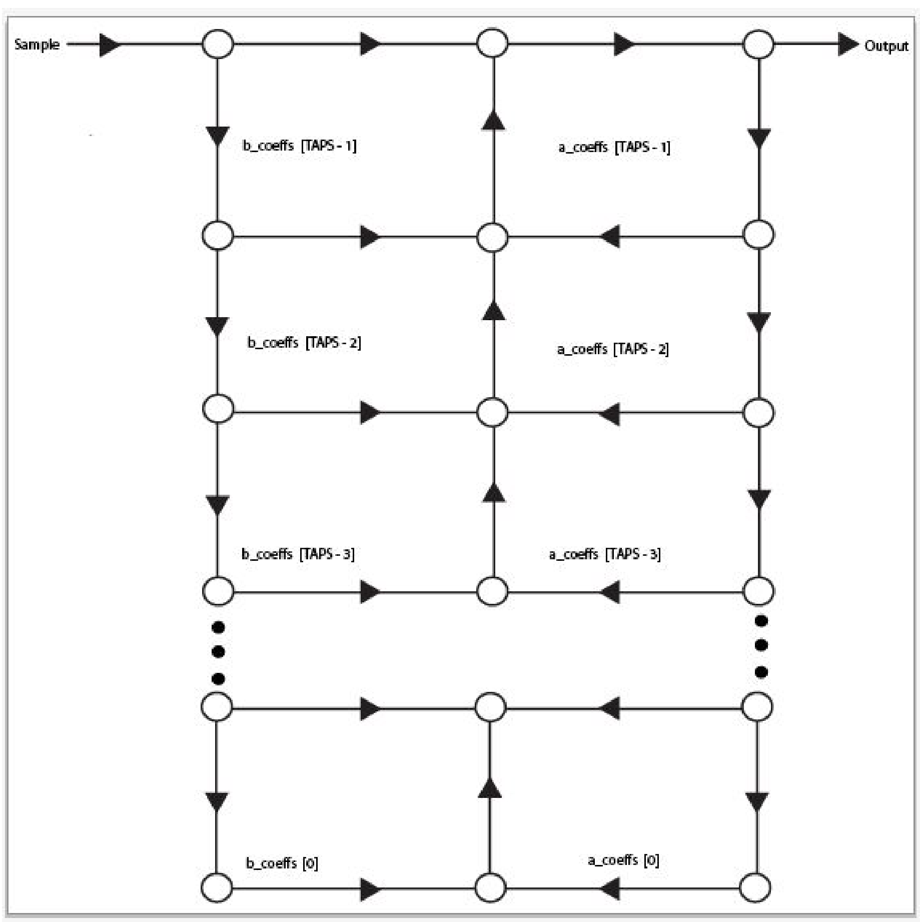
Each filter should have its own delay line which the function maintains in the array `state`. The array should be initialized to zero before calling the function for the first time and should not be modified by the calling program. The length of the `state` array should be `taps+1` as the function uses the array to store a pointer to the current delay line.

CAUTION: 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 or ADSP-21369 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-2136x processors specified above.

The flow graph (the ***Flow Graph*** figure) corresponds to the `iir()` routine as part of the DSP run-time library. The `b_coeffs` array should equal TAPS +1, while the `a_coeffs` array should equal TAPS.

The `biquad` function should be used instead of the `iir` function if a multi-stage filter is required.

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 bi-quad, 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. 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 re-started. 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:

$$H(z) = \prod_{n=0}^{\text{sections}-1} \frac{1 + \left(\frac{b_n 1}{b_n 0}\right) z^{-1} + \left(\frac{b_n 2}{b_n 0}\right) z^{-2}}{1 + \left(\frac{a_n 1}{a_n 0}\right) z^{-1} + \left(\frac{a_n 2}{a_n 0}\right) z^{-2}}$$

To get the correct amplitude of the signal, H(z) should be adjusted by this formula:

$$H(z) = H(z) \bullet \left(\prod_{n=0}^{\text{sections}-1} \frac{b_n 0}{a_n 0} \right)$$

Error Conditions

None

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;

   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 */

iir (input, output, coeffs, state, SAMPLES, SECTIONS);

/* Adjust output by all (b0/a0) terms */

for (i = 0; i < SAMPLES; i++)
   output[i] = output[i] * scale;
```

See Also

biquad, fir

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 functions

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);
```

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.

Error Conditions

None

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

NOTE: The `matmaddf` function (and `matmadd`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode 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);
```

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.

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

None

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

matmmlt, matmadd, matmsub, matsmlt

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);
```

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.

Error Conditions

None

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

matmsub, matmadd, matmmlt, matssub

NOTE: The `matmsubf` function (and `matmsub`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode 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

None

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

NOTE: The matsaddf function (and matsadd, if doubles are the same size as floats) uses SIMD by default. Refer to Implications of Using SIMD Mode 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);
```

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.

Error Conditions

None

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

matsmlt, matmmlt, matsadd, matssub

NOTE: The `matsmltf` function (and `matsmlt`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode 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

None

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

matssub, matmsub, matsadd, matsmlt

NOTE: The `matssubf` function (and `matssub`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode 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

None

Example

```
#include <stats.h>
#define SIZE 256

double data[SIZE];
double result;

result = mean (data, SIZE);
```

See Also

var

NOTE: The `meanf` function (and `mean`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode for more information.

mu_compress

µ-law compression

Synopsis (Scalar-Valued)

```
#include <comm.h>

int mu_compress (int x);
```

Synopsis (Vector-Valued)

```
#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 (-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.

NOTE: The vector versions of `mu_compress` uses serial port 0 to perform the compressing 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

None

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];
mu_compress (data, compressed, NSAMPLES);
```

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)

```
#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 (-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 (-law definition). The array returned contains linear 14-bit signed samples. These functions returns a pointer to the output data array.

NOTE: The vector versions of mu_expand uses serial port 0 to perform the compressing 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

None

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];

mu_expand (data, expanded_data, NSAMPLES);
```

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.

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,-4.0};
```

```
complex_double z;
z = norm(x);      /* z = (0.4472,-0.8944) */
```

See Also

No related functions

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 input argument `phase` for `polarf` must be in the domain [-1.647e6, 1.647e6] and for `polard` must be in the domain [-8.433e8, 8.433e8]. The functions return a complex value of zero if the specified phase is outside their domain.

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

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 will contain 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 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.

NOTE: The library also contains the rfftf_2 function. (For more information, see rfftf_2.) 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

None

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];
complex_float c_temp[FFTSIZE];
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

accel_cfft, accel_ifft, accel_rfft, cfft, fft_magnitude, ifft, rfftf_2, rfftN, twidfft

rfft_mag

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 the `rfft_mag` function and is provided for compatibility with previous versions of CCES.

NOTE: 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:

$$magnitude(z) = \frac{\sqrt{Re(a_z)^2 + Im(a_z)^2}}{fftsize}$$

Error Conditions

None

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, fft_magnitude, fftf_magnitude, rfftN

NOTE: By default, this function uses SIMD. Refer to Implications of Using SIMD Mode for more information.

rfftf_2

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 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.

NOTE: The function invokes the `cfftf` function, which has been implemented to make highly efficient use of the processor's SIMD capabilities. 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.
- Due to the alignment restrictions of the input arrays (as documented above), it is unlikely that the function will generate the correct results if the input arrays are allocated on the stack.

For more information, refer to Implications of Using SIMD Mode.

Error Conditions

None

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
static float dm fft1_re[FFT_SIZE];
static float dm fft1_im[FFT_SIZE];

#pragma align 64
static float pm fft2_re[FFT_SIZE];
static 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, fftf_magnitude, ifftf, rfft, rfftN, twidfftf

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.

NOTE: 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 rfftN instead.

Error Conditions

None

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, fftf_magnitude, ifftN, rfft, rfftN

rfftN

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 SHARC processors. 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.

NOTE: 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

None

Example

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

float input[N];
complex_float output[N/2];

rfft2048 (input, output);
```

See Also

cfftN, ifftN, rfft, rfftN, rfftf_2

NOTE: By default, these functions use SIMD. Refer to Implications of Using SIMD Mode 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

None

Example

```
#include <stats.h>
#define SIZE 256

double data[SIZE];
double result;

result = rms (data, SIZE);
```

See Also

mean, var

NOTE: The `rmsf` function (and `rms`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode 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

No related functions

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

None

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 functions

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 FFT's, 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

None

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

accel_twidfft, cfft, ifft, rfft, twidfftf

twidfftf

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.

NOTE: 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

None

Example

```
#include <filter.h>
#define FFT_SIZE 1024

#pragma section("seg_dmdata");
float twid_r[FFT_SIZE/2];
#pragma section("seg_pmdata")
float twid_i[FFT_SIZE/2];

#pragma align 1024
#pragma section("seg_dmdata")
float input_r[FFT_SIZE];
#pragma align 1024
#pragma section("seg_pmdata")
float input_i[FFT_SIZE];

#pragma section("seg_dmdata")
float temp_r[FFT_SIZE];
#pragma 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, fft_magnitude, rfftf_2, 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

None

Example

```
#include <stats.h>
#define SIZE 256

double data[SIZE];
double result;

result = var (data, SIZE);
```

See Also

mean

NOTE: The `varf` function (and `var`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode 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

None

Example

```
#include <vector.h>
#define N 100

double x[N], y[N];
double answer;

answer = vecdot (x, y, N);
```

See Also

cvecdot

NOTE: The `vecdotf` function (and `vecdot`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode 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

None

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

NOTE: The `vecsaddf` function (and `vecsadd`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode 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

None

Example

```
#include <vector.h>
#define N 100

double input[N], result[N];
double x;

vecsmlt (input, x, result, N);
```

See Also

vecsmlt, vecsadd, vecssub, vecvmlt

NOTE: The `vecsmltf` function (and `vecsmlt`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode 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

None

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

NOTE: The `vecssubf` function (and `vecssub`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode 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

None

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

NOTE: The `vecvaddf` function (and `vecvadd`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode 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

None

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

vecvmlt, vecsmlt, vecvadd, vecvsub

NOTE: The `vecvmltf` function (and `vecvmlt`, if `doubles` are the same size as `floats`) uses SIMD by default. Refer to Implications of Using SIMD Mode 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

None

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

NOTE: The vecvsubf function (and vecvsub, if doubles are the same size as floats) uses SIMD by default. Refer to Implications of Using SIMD Mode 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

None

Example

```
#include <stats.h>
#define SIZE 256

double input[SIZE];
int result;

result = zero_cross (input, SIZE);
```

See Also

No related functions

Index

D

Symbols

D

E

F

G

H

I

L

T

U

V

W

Z