CrossCore®
Embedded Studio

CCES 2.9.0 C/C++ Compiler and Library Manual for Blackfin Processors

(Includes Blackfin+ Processors)

**ANALOG
DEVICES**

Copyright Information

Contents

Optimal Performance from C/C++ Source Code

DSP Run-Time Library

Multicore Programming

Utility ROMs

1 Preface

Thank you for purchasing CrossCore® Embedded Studio (CCES), Analog Devices development software for Black-fin® embedded media processors.

Purpose

The *C/C++ Compiler and Library Manual* contains information about the C/C++ compiler and run-time libraries for Blackfin embedded processors that support a Media Instruction Set Computing (MISC) architecture. This architecture is the natural merging of RISC, media functions, and signal processing characteristics that delivers signal processing performance in a microprocessor-like environment.

Intended Audience

The primary audience for this manual are programmers who are familiar with Analog Devices Blackfin processors. This manual assumes that the audience has a working knowledge of the Blackfin processors' architecture and instruction set and C/C++ programming languages.

Programmers who are unfamiliar with Blackfin 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 Blackfin processor architecture and instructions).

Manual Contents

The manual consists of:

- Compiler provides information on compiler options, language extensions, C/C++/assembly interfacing, and support for C++ templates.

- Optimal Performance from C/C++ Source Code shows how to optimize compiler operation.

- C/C++ Run-Time Library shows how to use library functions and provides a complete C/C++ library function reference.

- DSP Run-Time Library shows how to use DSP library functions and provides a complete DSP library function reference.

- [Multicore Programming](#) provides various approaches and programming guidance for developing systems on dual-core Blackfin processors

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 name *Blackfin* refers to a family of 16-bit, embedded processors. Refer to the CCES online help for a complete list of supported processors. The name *Blackfin+*® refers to a family of processors with up to 400 MHz performance core and dual 16-bit or single 32-bit MAC support per cycle. 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.

Example	Description
`[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`.
`.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 *ATTEN-TION:* appears instead of this symbol.

2 Compiler

The C/C++ compiler (`ccblkfn`) is part of Analog Devices development software for Blackfin processors.

This chapter contains:

- C/C++ Compiler Overview, provides an overview of the C/C++ compiler for Blackfin processors.

- Compiler Command-Line Interface , describes the operation of the compiler as it processes programs, including input and output files and command-line switches.

- Using Native Fixed-Point Types, describes the compiler's support for the native fixed-point types `fract` and `accum`, defined in Chapter 4 of the "*Extensions to support embedded processors*" ISO/IEC draft technical report TR 18037.

- Language Standards Compliance, describes how to enable the best possible compliance to the ISO/IEC 9899:1990 C standard, the ISO/IEC 9899:1999 C standard, the ISO/IEC 14882:2003 C++ standard, or the ISO/IEC 14882:2011 C++ standard.

- MISRA-C Compiler, describes the compiler support for MISRA-C:2004 Guidelines for the use of the C language in critical systems.

- Run-Time Checking, describes the additional run-time checks supported by the compiler.

- C/C++ Compiler Language Extensions, describes the `ccblkfn` compiler's extensions to the ANSI/ISO standard for the C and C++ languages.

- C/C++ Preprocessor Features, contains information on the preprocessor and ways to modify source compilation.

- C/C++ Run-Time Model and Environment, contains reference information about implementation of C/C++ programs, data, and function calls in Blackfin processors.

- Compiler C++ Template Support, describes how templates are instantiated at compile time.

- File Attributes, describes how file attributes help with the placement of run-time library functions.

- Implementation Defined Behavior, describes how the compiler implements language features for which the standards allow some flexibility.

C/C++ Compiler Overview

The C/C++ compiler is designed to aid your DSP project development efforts by:

- Processing C and C++ source files, producing machine-level versions of the source code and object files

- Providing relocatable code and debugging information within the object files

- Providing relocatable data and program memory segments for placement by the linker in the processors' memory

Using C/C++, developers can significantly decrease time-to-market since it gives them the ability to efficiently work with complex signal processing data types. It also allows them to take advantage of specialized signal processing operations without having to understand the underlying processor architecture.

The C/C++ compiler compiles ANSI/ISO standard C and C++ code to support signal data processing. Additionally, Analog Devices includes within the compiler a number of C language extensions designed to assist in DSP development. The ccblkfn compiler runs from the CCES environment or from the operating system command line.

The C/C++ compiler processes your C and C++ language source files and produces Blackfin assembler source files. The assembler source files are assembled by the Blackfin processor assembler (easmblkfn). The assembler creates Executable and Linkable Format (ELF) object files that can be linked (using the linker) to create a Blackfin processor executable file or included in an archive library using the librarian tool (elfar). The way in which the compiler controls the assemble, link, and archive phases of the process depends on the source input files and the compiler options used.

Your source files contain the C/C++ program to be processed by the compiler. The ccblkfn compiler supports the following standards, each with Analog Devices extensions enabled:

- A hosted implementation of the ISO/IEC 9899:1990 C standard ("C89").

- A freestanding implementation of the ISO/IEC 9899:1999 C standard ("C99").

- A hosted implementation of the ISO/IEC 14882:2003 C++ standard ("C++ 2003").The compiler supports the language features supported by a standard subset of the C++ Library. You can view the abridged C++ library reference in the CCES online help.

- A freestanding implementation of the ISO/IEC 14882:2011 C++ standard ("C++ 2011").

RTTI and exceptions for C++ are supported, but disabled by default. For information on these switches see -rtti and -eh.

For information on the C or C++ language standards, see any of the many reference texts.

The ccblkfn compiler supports a set of C/C++ language extensions. These extensions support hardware features of the Blackfin processors. For information on these extensions, see C/C++ Compiler Language Extensions.

You can specify compiler options from the Preference pages of the CCES Integrated Development Environment (IDE). These selections control how the compiler processes your source files, letting you select features that include the language dialect, error reporting, and debugger output.

The Preferences pages are accessible from the *Properties* choice on the *Project* menu. Within the Preferences pages, navigate to *C/C++ Build*, then to *Settings*. Alternatively, click on the *Settings* icon in the Project Explorer view. For both routes, the compiler options are then available from *Settings > Tool Settings > CrossCore Blackfin C/C++ Compiler*.

For more information on the CCES environment, refer to the online help.

Compiler Components

The compiler is not a single program, but a collection of programs, each with a different task.

Compiler Driver

The compiler driver, `ccblkfn`, is the user interface to the other programs, and is the program you invoke when you run the compiler on the command line. Its responsibility is to marshal and interpret the command-line arguments to determine what other components and code-generation tools need invoking, and in what order. The compiler driver hides the complexity and presents a consistent interface. For this reason, throughout the documentation, "the compiler", "compiler driver" and "`ccblkfn`" are used interchangeably.

Compiler Proper

The compiler proper, found in `Blackfin\etc\compiler`, is the actual compiler; it compiles a single C/C++ source file into a single assembly output file. The compiler driver invokes the compiler proper for each C/C++ source file specified.

Assembler

The assembler, `easmblkfn`, assembles a single assembly source file into a single object file. The compiler driver invokes the assembler to translate both user-supplied assembly files and compiler-generated assembly files.

Linker

The linker, `linker`, combines object files into executable files, and searches library files to resolve references to undefined symbols. The linker relies on a `.ldf` file to specify how the resulting collection of symbols should be mapped into memory. The compiler driver invokes the linker when the specified output file is an executable file.

Prelinker

The prelinker is found at `Blackfin\etc\prelinker`. Its purpose is to examine the set of objects and libraries prior to linking, and to instruct the compiler driver to recompile files or add additional libraries or switches, as needed. The compiler driver invokes the prelinker just prior to invoking the linker. Language features supported by the prelinker include:

- C++ template instantiation
- Interprocedural Analysis
- Instrumented Profiling

IPA Solver

The IPA Solver, `Blackfin\etc\ipa`, propagates information between compiled modules, as part of Interprocedural Analysis. The IPA Solver might direct the compiler driver to recompile a source file, if propagated information can improve optimization. The IPA Solver is invoked by the prelinker when any of the input files were compiled with IPA optimization enabled.

PGO Merger

The PGO merger, `Blackfin\etc\pgo`, combines multiple profiles gathered through profiled executions of an application, and produces a single profile for the compiler to use. The PGO merger is invoked by the compiler driver whenever more than one PGO profile is specified.

Librarian

The librarian, `elfar`, provides facilities for creating, modifying and inspecting library files. The compiler driver invokes the librarian when the output file is a library file.

Memory Initializer

The memory initializer, `meminit`, creates an initialization stream within the executable file. The compiler driver directs the linker to invoke the memory initializer after linking, when the -mem switch is specified.

NOTE: The assembler, linker and librarian are documented in the *Assembler and Preprocessor Manual* and *Linker and Utilities Manual*. The other components should always be invoked only through the compiler driver, never directly.

Compiler Command-Line Interface

This section describes how the `ccblkfn` compiler is invoked from the command line, the various types of files used by and generated from the compiler, and the switches used to tailor the compiler's operation

This section contains:

- Running the Compiler
- C/C++ Compiler Command-Line Switches
- Environment Variables Used by the Compiler
- Additional Path Support

- Optimization Control

- Controlling Silicon Revision and Anomaly Workarounds Within the Compiler

By default, the compiler runs with Analog extensions for C code enabled. This means that the compiler processes source files written in ISO/IEC 9899:1999 standard C language supplemented with Analog Devices extensions. The *File Extensions Specifying Compiler Action* table (in Running the Compiler) lists valid extensions of source files the compiler operates upon. By default, the compiler processes input files through the listed stages to produce a `.dxe` file. See file names in the *Input and Output File Extensions* table (in Running the Compiler).

The *C/C++ Mode Selection Switches* table (in C/C++ Compiler Command-Line Switches) lists switches that select the language dialect. Although many switches are generic between C and C++, some are valid in C++ mode only. A summary of the generic C/C++ compiler switches appears in the *C/C++ Compiler Common Switches* table (in C/C+ + Compiler Command-Line Switches). A summary of the C++-specific compiler switches appears in the *C Mode (MISRA) Compiler Switches* table (in C/C++ Compiler Command-Line Switches). The summaries are followed by descriptions of each switch.

NOTE: When developing a DSP project, sometimes it is useful to modify the compiler's default options settings. The way the compiler's options are set depends on the environment used to run the DSP development software. For more information, see Environment Variables Used by the Compiler.

Running the Compiler

Use the following syntax for the `ccblkfn` command line:

`ccblkfn [-switch [-switch]] sourcefile [sourcefile]`

The following table describes the command-line syntax.

Table 2-1: ccblkfn Command-Line Syntax

Parameter	*Description*
`ccblkfn`	Name of the compiler program for Blackfin processors.
`-switch`	Switch (or switches) to process. The compiler has many switches. These switches select the operations and modes for the compiler and other tools. Command-line switches are case-sensitive. For example, -O is not the same as -o.
`sourcefile`	Name of the file to be preprocessed, compiled, assembled, and/or linked. The *sourcefile* can include the directory, file name, and file extension. The compiler supports both Win32-style and POSIX-style paths, using either forward slashes or back slashes as the directory delimiter. It also supports UNC path names (starting with two slashes and a network name).

NOTE: When file names or other switches for the compiler include spaces or other special characters, ensure that these are properly quoted (usually using double-quote characters), to ensure that they are not interpreted by the operating system before being passed to the compiler.

The `ccblkfn` compiler uses the file extension to determine what the file contains and what operations to perform upon it. The *Input and Output File Extensions* table lists the allowed extensions.

Table 2-2: Input and Output File Extensions

File Extension	File Extension Description
`.c .C`	C source file
`.cpp .cxx .cc .c++`	C++ source file
`.h`	Header file (referenced by an `#include` statement)
`.hpp .hh .hxx .h++`	C++ header file (referenced by a `#include` statement)
`.hpl`	Heap debugging output file-used by the Reporter Tool to produce a report on heap usage and related errors
`.ii .ti`	Template instantiation files-used internally by the compiler when instantiating templates
`.et`	Exported template files-used internally by the compiler when instantiating exported templates
`.ipa`	Interprocedural analysis files-used internally by the compiler when performing interprocedural analysis.
`.pgo .pgi .pgt`	Execution profile generated by a simulation run or instrumented executable
`.i`	Preprocessed source file-created when preprocess only is specified
`.s .asm`	Assembly language source files
`.is`	Preprocessed assembly language source-retained when -save-temps is specified.
`.sbn`	Binary data included by an assembly language source file
`.ldf`	Linker description file
`.misra`	Text file used by prelinker for MISRA-C Guidelines checking
`.doj .o`	Object file to be linked
`.dlb .a`	Library of object files to be linked as needed
`.dxe`	Executable file produced by compiler
`.xml`	Processor memory map file output
`.sym`	Processor symbol map file output

Example 1.

The following command line runs `ccblkfn` with the following options:

```
ccblkfn -proc ADSP-BF533 -O -Wremarks -o program.dxe source.c
```

where:

-proc ADSP-BF533

>   Specifies compiler instructions unique to the ADSP-BF533 processor

-O

> Specifies optimization for the compiler

-Wremarks

> Selects extra diagnostic remarks in addition to warning and error messages

-o program.dxe

> Specifies a name for the compiled, linked output

source.c

> Specifies the C language source file to be compiled

Example 2.

The following command line for C++ mode runs `ccblkfn` with these options:

```
ccblkfn -proc ADSP-BF533 -c++ source.cpp
```

where:

-c++

> Specifies all of the source files to be compiled in C++ mode

source.cpp

> Specifies the C++ language source file to be compiled

The normal function of `ccblkfn` is to invoke the compiler, assembler, and linker as required to produce an executable object file. The precise operation is determined by the extensions of the input file names and by various switches.

In normal operation, the compiler uses the files listed in the *File Extensions Specifying Compiler Action* table to perform a specified action.

Table 2-3: File Extensions Specifying Compiler Action

Extension	*Action*
`.c .C .cpp .cxx .cc .c++`	Source file is compiled, assembled, and linked.
`.asm .dsp .s`	Assembly language source file is assembled and linked.
`.doj`	Object file (from previous assembly) is linked.
`.pgo .pgi`	Profile-guided optimization information file is used during compilation.

If multiple files are specified, each is processed to produce an object file and then all the object files are presented to the linker.

You can stop this sequence at various points using appropriate compiler switches (`-E`, `-P`, `-M`, `-H`, `-S`, and `-c`), or by selecting options within the IDE.

Many of the compiler's switches take a file name as an optional parameter. If you do not use the optional output name switch, `ccblkfn` names the output for you. The *Input and Output File Extensions* table lists the type of files, names, and extensions `ccblkfn` appends to output files.

File extensions vary by command-line switch and file type. These extensions are influenced by the program that is processing the file. The programs search directories that you specify and path information that you include in the file name. The *Input and Output File Extensions* table indicates the extensions that the preprocessor, compiler, assembler, and linker support. The compiler supports relative and absolute directory names to define file extension paths. For information on additional search directories, see the command-line switch that controls the specific type of extensions.

When providing an input or output file name as an optional parameter, follow these guidelines.

- Use a file name (include the file extension) with an unambiguous relative path or an absolute path. A file name with an absolute path includes the directory, file name, and file extension. The compiler uses the file extension convention listed in the *Input and Output File Extensions* table to determine the input file type.

- Verify that the compiler is using the correct file. If you do not provide the complete file path as part of the parameter or add additional search directories, `ccblkfn` looks for input in the current directory.

   NOTE: Use the verbose output switches for the preprocessor, compiler, assembler, and linker to cause each of these tools to display command-line information as they process each file.

The compiler refers to a number of environment variables during its operation, and these environment variables can affect the compiler's behavior. Refer to Environment Variables Used by the Compiler for more information.

C/C++ Compiler Command-Line Switches

This section describes command-line switches used when compiling. Tables, organized by switch type, provide a brief description of each switch. Following these tables is a detailed description of each switch.

This section contains the following tables:

- C/C++ Mode Selection Switch Descriptions

- C/C++ Compiler Common Switch Descriptions

- C Mode (MISRA) Compiler Switch Descriptions

- C++ Mode Compiler Switch Descriptions

Table 2-4: C/C++ Mode Selection Switches

Switch	Description
-c89	Supports programs that conform to a hosted implementation of the ISO/IEC 9899:1990 standard with Analog Devices extensions
-c99	Supports programs that conform to a freestanding implementation of the ISO/IEC 9899:1999 standard with Analog Devices extensions. This is the default mode.
-c++	Supports programs that conform to a hosted implementation of the ISO/IEC 14882:2003 C++ standard with Analog Devices extensions (see -full-cpplib).
-c++11	Supports programs that conform to an implementation of the ISO/IEC 14882:2011 standard with Analog Devices language extensions.
-g++	Supports programs that conform to an implementation of the GNU C++. By default the C++ dialect is based on the ISO/IEC 14882:2003 standard.

Table 2-5: C/C++ Compiler Common Switches

Switch	Description
sourcefile	This parameter specifies the file to be compiled.
-@ filename	Reads command-line input from the file.
-A name (tokens)	Asserts the specified name as a predicate.
-absolute-path-dependencies	Uses absolute paths for the make dependencies emitted when using the −M, −MM or −MD switches.
-add-debug-libpaths	Links against debug-specific variants of system libraries, where available.
-alttok	Allows alternative keywords and sequences in sources.
-always-inline	Treats `inline` keyword as a requirement rather than a suggestion.
-annotate	Enables assembly annotations.
-annotate-loop-instr	Provides additional annotation information for the prolog, kernel and epilog of a loop.
-auto-attrs	Directs the compiler to emit automatic attributes based on the files it compiles. Enabled by default.
-bss	Causes the compiler to put global zero-initialized data into a separate BSS-style section. Set by default.
-build-lib	Directs the librarian to build a library file.
-C	Retains preprocessor comments in the output file.
-c	Compiles and/or assembles only, but does not link.
-component file.xml	Reads additional options from the specified XML file.
-const-read-write	Specifies that data accessed via a pointer to `const` data may be modified elsewhere.
-const-strings	Directs the compiler to mark string literals as `const` qualified.
-cplbs	Instructs the compiler to assume that CPLBs are active.

Table 2-5: C/C++ Compiler Common Switches (Continued)

Switch	Description
-D *macro*[=*definition*]	Defines `macro`.
-dcplbs	Instructs the compiler to assume that data CPLBs are active.
-decls-{weak|strong}	Determines whether uninitialized global variables should be treated as definitions or declarations.
-dependency-add-target target	Adds target to any emitted dependency information.
-double-size-{32 | 64} `-double-size-64`	Selects 32- or 64-bit IEEE format for `double.-double-size-32` is the default mode.
-double-size-any	Indicates that the resulting object can be linked with objects built with any `double` size.
-dry	Displays, but does not perform, main driver actions (verbose dry run).
-dryrun	Displays, but does not perform, top-level driver actions (terse dry run).
-E	Preprocesses, but does not compile, the source file.
-ED	Preprocesses and sends all output to a file.
-EE	Preprocesses and compiles the source file.
-eh	Enables exception handling.
-enum-is-int	By default, an `enum` can have a type larger than `int`. This option ensures the `enum` type is `int`.
-expand-symbolic-links	Provides support for Cygwin path extensions within command-line switches and `#include` preprocessor directives.
-expand-windows-shortcuts	Provides support for Windows shortcuts within command-line switches and `#include` preprocessor directives.
-extra-keywords	Recognizes Blackfin processor extensions to ANSI/ISO standards for C (default mode).
-file-attr *name[=value]*	Adds the specified attribute *name/value* pair to the file(s) being compiled.
-fixed-point-io	Links with a variant of the Analog Devices I/O library containing support for printing native fixed-point types in decimal format.

-flags{-asm | -compiler | -ipa | -lib | -link | -mem | -prelink} switch[, switch2[,...]]
`-flags-compiler switches`
`-flags-ipa switches`
`-flags-lib switches`
`-flags-link switches`
`-flags-mem switches`
`-flags-prelink switches`
	Passes command-line switches through the compiler to other build tools.

-force-circbuf	Treats array references of the form `array[i%n]` as circular buffer operations.

Table 2-5: C/C++ Compiler Common Switches (Continued)

Switch	Description
-force-link	Forces stack frame creation for leaf functions. Defaults to ON with `-g` option set, enforced for the `-p` option.
-fp-associative	Treats floating-point multiplication and addition as associative operations.
-full-io	Links with a third party, proprietary I/O library.
-full-version	Displays the version number of the driver and processes invoked by the driver.
-fx-contract	Sets the default mode of `FX_CONTRACT` to `ON`.
-fx-rounding-mode-biased	Sets the default mode of `FX_ROUNDING_MODE` to `BIASED`.
-fx-rounding-mode-truncation	Sets the default mode of `FX_ROUNDING_MODE` to `TRUNCATION`.
-fx-rounding-mode-unbiased	Sets the default mode of `FX_ROUNDING_MODE` to `UNBIASED`.
-g	Generates DWARF-2 debug information.
-glite	Generates lightweight DWARF-2 debug information.
-gnu-style-dependencies	Produces dependency information in the style expected by the GNU `make` program.
-H	Outputs a list of included header files, but does not compile.
-HH	Outputs a list of included header files and compiles.
-h[elp] -help	Outputs a list of command-line switches with brief syntax descriptions.
-I `directory[{,|;} directory...]`	Appends `directory` to the standard search path.
-I-	Specifies the point in the `include` directory list where the search for header files enclosed in angle brackets should begin.
-i	Outputs only header details or makefile dependencies for `include` files specified in double quotes.
-icplbs	Instructs the compiler to assume that instruction CPLBs are active.
-include `filename`	Includes named file prior to each source file.
-ipa	Specifies that interprocedural analysis should be performed for optimization between translation units.
-jcs2l	Enables the conversion of `short` jumps to `long` jumps when necessary but uses the `P1` register for indirect jumps when long jumps are insufficient (enabled by default).
-L `directory[{,|;} directory]`	Appends `directory` to the standard library search path.
-l `library`	Searches `library` for functions when linking.
-list-workarounds	Lists all compiler-supported errata workarounds.
-M	Generates `make` rules only, but does not compile.
-MD	Generates `make` rules, compiles, and prints to a file.

Table 2-5: C/C++ Compiler Common Switches (Continued)

Switch	Description
-MM	Generates `make` rules and compiles.
-Mo `filename`	Writes dependency information to `filename`. This switch is used in conjunction with the `-ED` or `-MD` options.
-Mt `name`	Makes dependencies, where the target is renamed as `filename`.
-map `filename`	Directs the linker to generate a memory map of all symbols.
-mem	Causes the compiler to invoke the Memory Initializer after linking the executable file.
-multiline	Enables string literals over multiple lines (default).
-never-inline	Ignores `inline` keyword on function definitions.
-no-alttok	Disallows alternative keywords and sequences in sources.
-no-annotate	Disables the annotation of assembly files.
-no-annotate-loop-instr	Disables the production of additional loop annotation information by the compiler (default mode).
-no-assume-vols-are-mmrs	Directs the compiler not to apply workarounds for MMR-related silicon errata to arbitrary `volatile`-qualified memory accesses.
-no-auto-attrs	Directs the compiler not to emit automatic attributes based on the files it compiles.
-no-bss	Causes the compiler to group global zero-initialized data into the same section as global data with non-zero initializers.
-no-circbuf	Disables the automatic generation of circular buffering code.
-no-const-strings	Directs the compiler not to make string literals `const` qualified.
-no-cplbs	Directs the compiler that CPLBs are not enabled.
-no-defs	Disables preprocessor definitions: macros, include directories, library directories or keyword extensions.
-no-eh	Disables exception-handling.
-no-expand-symbolic-links	Disables support for Cygwin path extensions in command-line paths and preprocessor include directives.
-no-expand-windows-shortcuts	Disables support for Windows shortcuts in command-line paths and preprocessor include directives.
-no-extra-keywords	Disables language extension keywords that could be valid C/C++ identifiers.
-no-force-link	Does not create a new stack frame for leaf functions, if one can be omitted. Overrides the default for `-g`.
-no-fp-associative	Does not treat floating-point multiplication and addition as associative operations.
-no-full-io	Links with the Analog Devices I/O library; enabled by default.
-no-fx-contract	Sets the default mode of `FX_CONTRACT` to `OFF`.
-no-int-to-fract	Prevents the compiler from turning integer into fractional arithmetic

Table 2-5: C/C++ Compiler Common Switches (Continued)

Switch	Description
-no-jcs2l	Prevents the linker from converting compiler-generated short jumps to long jumps using register P1.
-no-mem	Causes the compiler to not invoke the Memory Initializer after linking; set by default.
-no-multiline	Disables multiple line string literal support.
-no-progress-rep-timeout	Prevents the compiler from issuing a diagnostic during excessively long compilations.
-no-rtcheck	Disables run-time checking.
-no-rtcheck-arr-bnd	Disables checking of array boundaries at run-time.
-no-rtcheck-div-zero	Disables checking for division by zero at run-time.
-no-rtcheck-heap	Disables checking of heap operations at run-time.
-no-rtcheck-null-ptr	Disables checking for NULL pointer dereferences at run-time.
-no-rtcheck-shift-check	Disables checking for negative/too-large shifts at run-time.
-no-rtcheck-stack	Disables checking for stack overflow at run-time.
-no-rtcheck-unassigned	Disables checking for unassigned variables at run-time.
-no-sat-associative	Saturating addition is not associative.
-no-saturation	Causes the compiler not to introduce saturation semantics when optimizing expressions that do not explicitly specify saturating semantics.
-no-std-ass	Prevents the compiler from defining standard assertions.
-no-std-def	Disables normal macro definitions and also Analog Devices keyword extensions that do not have leading underscores (__).
-no-std-inc	Searches only for preprocessor `include` header files in the current directory and in directories specified with the `-I` switch.
-no-std-lib	When linking, searches for only those library files specified with the `-l` switch.
-no-threads	Specifies that no support is required for multi-threaded applications
-no-utility-rom	Do not link against the Utility ROM (ADSP-BF592-A and ADSP-BF7xx family processors only).
-no-workaround *workaround_id[, workaround_id]*	Disables specific hardware anomaly workarounds within the compiler.
-no-zero-loop-counters	Do not zero loop counters (`LC0` and `LC1`) on function exit.
-O[0|1] -O1 -O0	Enables (`-O` or `-O1`) or disables (`-O0`) code optimizations (uppercase "O" optionally followed by a zero or a one).
-Oa	Enables automatic function inlining.
-Os	Optimizes the file to decrease code size.

Table 2-5: C/C++ Compiler Common Switches (Continued)

Switch	Description
-Ov *num*	Controls speed versus size optimizations.
-o *filename*	Specifies the output file name.
-overlay	Disables the propagation of register information between functions and forces the compiler to assume that all functions clobber all scratch registers
-overlay-clobbers *clobbered-regs*	Specifies the registers assumed to be clobbered by an overlay manager.
-P	Preprocesses, but does not compile, the source file; output does not contain #line directives.
-PP	Preprocesses and compiles the source file; output does not contain #line directives.
-p	Generates profiling instrumentation.
-path {-asm | -compiler | -ipa | -lib | -link | -prelink} *pathname* -path-compiler *filename* -path-ipa *filename* -path-lib *filename* -path-link *filename* -path-prelink *filename*	Uses the specified filename as the location of the specified compilation tool (assembler, compiler, IPA solver, library builder, linker or prelinker).
-path-install *directory*	Uses the specified directory as the location of all compilation tools.
-path-output *directory*	Specifies the location of non-temporary files.
-path-temp *directory*	Specifies the location of temporary files.
-pgo-session *session-id*	Used with profile-guided optimization.
-pguide	Adds instrumentation for the gathering of a profile as the first stage of performing profile-guided optimization.
-pplist *filename*	Outputs a raw preprocessed listing to the specified file.
-proc *processor*	Specifies a processor for which the compiler should produce suitable code.
-prof-hw	Instructs the compiler to generate profiling code targeted for execution on hardware. Requires use of a supported profiling switch.
-progress-rep-func	Issues a diagnostic message each time the compiler starts compiling a new function. Equivalent to -Wwarn=cc1472.
-progress-rep-opt	Issues a diagnostic message each time the compiler starts a new optimization pass on the current function. Equivalent to -Wwarn=cc1473.
-progress-rep-timeout	Issues a diagnostic message if the compiler exceeds a time limit during compilation.
-progress-rep-timeout-secs *secs secs*	Specifies how many seconds must elapse during a compilation before the compiler issues a diagnostic on the length of compilation.
-R *directory[, directory]*	Appends *directory* to the standard search path for source files.
-R-	Removes all directories from the source file search directory list.

Table 2-5: C/C++ Compiler Common Switches (Continued)

Switch	Description
-reserve *register[, register]*	Reserves certain registers from compiler use. **NOTE:** Reserving registers can have a detrimental effect on the compiler's optimization capabilities.
-rtcheck	Enables run-time checking.
-rtcheck-arr-bnd	Enables checking of array boundaries at run-time.
-rtcheck-div-zero	Enables checking for division by zero at run-time
-rtcheck-heap	Enables checking of heap operations at run-time.
-rtcheck-null-ptr	Enables checking for NULL pointer dereferences at run-time.
-rtcheck-shift-check	Enables checking for negative/too-large shifts at run-time.
-rtcheck-stack	Enables checking for stack overflow at run-time.
-rtcheck-unassigned	Enables checking for unassigned variables at run-time.
-S	Stops compilation before running the assembler.
-s	When linking, removes debugging information from the output executable file.
-sat-associative	Saturating addition is associative.
-save-temps	Saves intermediate files.
-sdram	Instructs the compiler to assume that at least bank 0 of external SDRAM will be present and enabled.
-section *id=section_name[, id=section_name...]* =section	Orders the compiler to place data/program of type *id* into the section *section*.
-show	Displays the driver command-line information.
-signed-bitfield	Makes the default type for `int bitfields` signed.
-signed-char	Makes the default type for `char` signed.
-si-revision *version*	Specifies a silicon revision of the specified processor. The default setting is the latest silicon revision at the time of release.
-structs-do-not-overlap	Specifies that `struct` copies may use `memcpy` semantics, rather than the usual `memmove` behavior.
-syntax-only	Checks the source code for compiler syntax errors, but does not write any output.
-sysdefs	Instructs the driver to define preprocessor macros that describe the current user and machine.
-T *filename*	Specifies the linker description file.
-threads	Enables the support for multi-threaded applications.
-time	Displays the elapsed time as part of the output information on each part of the compilation process.
-U *macro*	Undefines *macro*.

Table 2-5: C/C++ Compiler Common Switches (Continued)

Switch	Description
-unsigned-bitfield	Makes the default type for plain `int` bit-fields unsigned.
-unsigned-char	Makes the default type for `char` unsigned.
-utility-rom	Link against the ROM (ADSP-BF592-A and ADSP-BF7xx processors only).
-v	Displays version and command-line information for all compilation tools.
-verbose	Displays command-line information for all compilation tools as they process each file.
-version	Directs the compiler to display its version number.

-W{annotation|error|remark|suppress|warn} `number[, number...]`
`-Werror number`
`-Wremark number`
`-Wsuppress number`
`-Wwarn number`
	Overrides the default severity of the specified messages (annotations, errors, remarks, or warnings).

-Wannotations	Indicates that the compiler may issue code generation annotations, which are messages milder than warnings that may help you to optimize your code.
-Werror-limit `number`	Stops compiling after reaching the specified number of errors.
-Werror-warnings	Directs the compiler to treat all warnings as errors.
-Wremarks	Issues compiler remarks.
-Wterse	Issues the briefest form of compiler warnings, errors, and remarks.
-w	Disables all warnings.
-warn-component	Issues warnings if any libraries specified by component XML files could not be located.
-warn-protos	Issues warnings about functions without prototypes.
-workaround `workaround_id[,` `workaround_id]`	Enables code generator workaround for specific hardware errata.
-xref `filename`	Outputs cross-reference information to the specified file.
-zero-loop-counters	Ensures used loop counters (`LC0` and `LC1`) are zeroed on function exit.

Table 2-6: C Mode (MISRA) Compiler Switches

Switch	Description
-misra	Enables checking for MISRA-C:2004 Guidelines. Allows some relaxation of interpretation. For more information, see Rule Descriptions.
-misra-linkdir `directory`	Specifies directory for generation of `.misra` files. If this option is not specified, a local directory called `MISRARepository` is created. The `.misra` files allow the compiler to record information across modules to support the implementation of MISRA rules 5.5, 8.8, and 8.10.

Table 2-6: C Mode (MISRA) Compiler Switches (Continued)

Switch	Description
-misra-no-cross-module	Implies −misra, but inhibits the generation of .misra files to check for link-time rule violations. It therefore disables checking of MISRA rules 5.5, 8.8, and 8.10.
-misra-no-runtime	Implies −misra, but inhibits the generation of extra code to perform run-time checking in support of Rule 21. The disabling of run-time checks also suppresses checking for rules 17.1, 17.2 and 17.3. It limits rules 9.1, 12.8, 16.3 and 17.4 to compile-time checks.
-misra-strict	Enables checking for MISRA-C:2004 Guidelines. Rules relaxed by -misra option are enforced fully by this option. For more information, see Rule Descriptions.
-misra-suppress-advisory	Implies −misra, but suppresses the reporting of advisory rules.
-misra-testing	Implies −misra, but suppresses reporting of MISRA rules 20.4, 20.7, 20.8, 20.9, 20.10, 20.11, and 20.12. This allows the use of I/O and other support functions during development testing.
-Wmis_suppress *rule_number[, rule_number]*	Overrides the default severity of the specified messages relating to the specified MISRA rules. For example, -Wmis_suppress 16.1 will suppress the reporting of violations of rule 16.1.
-Wmis_warn *rule_number[, rule_number]*	Overrides the default severity of the specified messages relating to the specified MISRA rules. For example, -Wmis_warn 16.1 will change the reporting of violations of rule 16.1 as an error to a warning.

Table 2-7: C++ Mode Compiler Switches

Switch	Description
-anach	Supports some language features (anachronisms) that are prohibited by the C++ standard but still in common use.
-check-init-order	Adds run-time checking to the generated code highlighting potential uninitialized external objects. For development purposes only-do not use in production code.
-friend-injection	Allows non-standard behavior with respect to friend declarations. When friend names are not injected, function names are visible only when using dependent lookup.
-full-cpplib	Directs the compilation to include ISO/IEC 14882:2003 C++ standard header files and link with the full standard library.
-full-dependency-inclusion	Ensures re-inclusion of implicitly included files when generating dependency information.
-implicit-inclusion	Allows implicit inclusion of source files as a method of finding definitions of template entities to be instantiated. It is not compatible with exported templates.
-no-anach	Disallows the use of anachronisms that are prohibited by the C++ standard.
-no-full-cpplib	Links the application with the abridged C++ library.
-no-friend-injection	Allows standard behavior. Friend function names are visible only when using argument-dependent lookup and friend class names are never visible. This is the default mode.
-no-implicit-inclusion	Prevents implicit inclusion of source files as a method of finding definitions of template entities to be instantiated. This is the default mode.

Table 2-7: C++ Mode Compiler Switches (Continued)

Switch	*Description*
-no-rtti	Disables run-time type information.
-no-std-templates	Disables the special lookup of names used in templates.
-rtti	Enables run-time type information.
-std-templates	Enables the lookup of names used in templates.

C/C++ Mode Selection Switch Descriptions

The following command-line switches provide C/C++ mode selection.

-c89

The −c89 switch directs the compiler to support programs that conform to the ISO/IEC 9899:1990 standard. For greater conformance to the standard, see Language Standards Compliance.

-c99

The −c99 switch directs the compiler to support programs that conform to a freestanding implementation of the ISO/IEC 9899:1999 standard. For greater conformance to the standard, see Language Standards Compliance.

> **NOTE:** The compiler supports the _Complex keyword but not the _Imaginary keyword.

-c++

The −c++ (C++ mode) switch directs the compiler to assume that the source file(s) are written in ANSI/ISO 14882:2003 standard C++ with Analog Devices language extensions. The compiler implicitly adds this switch when compiling files with a .cpp extension.

All the standard features of C++ are accepted in the default mode except exception handling and run-time type identification because these impose a run-time overhead that is not desirable for all embedded programs. Support for these features can be enabled with the -eh and -rtti switches. For greater conformance to the standard, see Language Standards Compliance.

-c++11

The −c++11 (C++ mode) switch directs the compiler to assume that the source file(s) are written in ANSI/ISO 14882:2011 standard C++ with Analog Devices language extensions.

This version of the compiler accepts many the features of the ANSI/ISO 14882:2011 standard, but the underlying library support conforms to the ANSI/ISO 14882:2003. Exception handling and run-time type identification have to be enabled explicitly, because these impose a run-time overhead that is not desirable for all embedded programs. Support for these features can be enabled with the -eh and -rtti switches. For greater conformance to the standard, see Language Standards Compliance.

-g++

The −g++ (C++ mode) switch directs the compiler to emulate a large number of extensions available in GNU C++. By default, the C++ dialect is based on the ANSI/ISO 14882:2003 standard. This dialect can be changed by additionally enabling the −c++11 switch.

C/C++ Compiler Common Switch Descriptions

The following command-line switches apply in both C and C++ modes.

sourcefile

The *sourcefile* parameter (or parameters) specifies the name of the file (or files) to be preprocessed, compiled, assembled, and/or linked. A file name can include the drive, directory, file name, and file extension. The `ccblkfn` compiler uses the file extension to determine the operations to perform. The *Input and Output File Extensions* and *File Extensions Specifying Compiler Action* tables (in Running the Compiler) list the permitted extensions and matching compiler operations.

-@ *filename*

The `-@` *filename* (command file) switch directs the compiler to read command-line input from *filename*. The specified file must contain driver options and may also contain source file names and environment variables. It can be used to store frequently used options as well as to read from a file list.

-A *name (tokens)*

The `-A` (assert) switch directs the compiler to assert *name* as a predicate with the specified *tokens*. This has the same effect as the `#assert` preprocessor directive. The following assertions (*Predefined Assertions* table) are predefined.

Table 2-8: Predefined Assertions

Assertion	*Value*
system	`embedded`
machine	`adspblkfn`
cpu	`adspblkfn`
compiler	`ccblkfn`

The `-A name(value)` switch is equivalent to including

```
#assert name(value)
```

in your source file, and both may be tested in a preprocessor condition in the following manner:

```
#if #name(value)
    // do something
#else
    // do something else
#endif
```

For example, the default assertions may be tested as:

```
#if #machine(adspblkfn)
    // do something else
#endif
```

> **NOTE:** The parentheses in the assertion need quotes when using the `-A` switch to prevent misinterpretation. Quotes are not required for an `#assert` directive in a source file.

-absolute-path-dependencies

The `-absolute-path-dependencies` switch can be used in conjunction with one of the `-M`, `-MM` or `-MD` switches. By default these switches emit make dependencies using relative paths. The -absolute-path-dependencies switch may be used to emit the dependencies using absolute rather than relative paths. The compiler driver will invoke all underlying tools (compiler, assembler, preprocessor, linker and archiver) with the switch as required so that all tools emit dependencies using absolute paths.

-add-debug-libpaths

The `-add-debug-libpaths` switch prepends the Debug subdirectory to the search paths passed to the linker. The Debug subdirectory, found in each of the silicon-revision-specific library directories, contains variants of certain libraries (for example, system services), which provide additional diagnostic output to assist in debugging problems arising from their use.

NOTE: Invoke this switch from the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Linker > Processor > Use Debug System libraries*.

-alttok

In C89 and C99 modes, the `-alttok` (alternative tokens) switch directs the compiler to allow digraph sequences in source files. This switch is enabled by default in C89 and C99 modes.

In C++ mode, this switch is disabled by default. When enabled in C++ mode, the switch also enables the recognition of alternative operator keywords listed in the *Alternative Operator Keywords* table, in C++ source files.

Table 2-9: Alternative Operator Keywords

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

NOTE: The `-alttok` switch has no effect on the use of the alternative tokens listed in the *Alternative Operator Keywords* table when in C89 or C99 mode. Instead, when in C89 or C99 mode, include header file `<iso646.h>` to use alternative tokens.

See also -no-alttok.

-always-inline

The `-always-inline` switch instructs the compiler to attempt to inline any call to a function that is defined with the `inline` qualifier. This switch is equivalent to applying `#pragma always_inline` to all functions in the module that have the `inline` qualifier.

See also -never-inline.

> **NOTE:** Invoke this switch from the IDE by setting *Project > Properties > C/C++ Build > Settings > Tool Settings > CrossCore Blackfin C/C++ Compiler > General > Inlining to All functions declared inline*.

-annotate

The `-annotate` (enable assembly annotations) switch directs the compiler to annotate assembly files generated by the compiler. By default, when optimizations are enabled, all assembly files generated by the compiler are annotated with information on the performance of the generated assembly. See Assembly Optimizer Annotations in the `Achieving Optimal Performance From C/C++ Source Code` chapter for more information on this feature.

> **NOTE:** Invoke this switch from the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > General > Generate annotations*.

See also -no-annotate.

-annotate-loop-instr

The `-annotate-loop-instr` switch directs the compiler to provide additional annotation information for the prolog, kernel, and epilog of a loop. See *Assembly Optimizer Annotations* for more details on this feature.

See also -no-annotate-loop-instr.

-auto-attrs

The `-auto-attrs` (automatic attributes) switch directs the compiler to emit automatic attributes based on the files it compiles. Emission of automatic attributes is enabled by default. See File Attributes for more information about attributes and what automatic attributes the compiler emits.

See also -no-auto-attrs and -file-attr *name[=value]* .

> **NOTE:** Invoke this switch from the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > CrossCore Blackfin C/C++ Compiler > General > Auto-generated attributes*.

-bss

The `-bss` switch causes the compiler to place global zero-initialized data into a BSS-style section (called `bsz`), rather than into the normal global data section. This is the default mode.

See also -no-bss.

-build-lib

The `-build-lib` (build library) switch directs the compiler to use `elfar` (the librarian) to produce a library file (`.dlb`) instead of using the linker to produce an executable file (`.dxe`). The `-o filename` switch must be used to specify the name of the resulting library.

-C

The `-C` (comments) switch, which is only active when used with the `-E`, `-ED`, `-P`, or `-PP` switches, directs the preprocessor to retain comments in its output.

-c

The `-c` (compile only) switch directs the compiler to compile and/or assemble the source files, but to stop before linking. The output is an object file (`.doj`) for each source file.

-component file.xml

The `-component` (read component file) switch instructs the compiler to read the specified XML file, and to retrieve additional switches for use when building applications that make use of the component. The IDE uses this switch to build projects that employ additional products beside CCES.

See also -warn-component.

-const-read-write

The `-const-read-write` switch directs the compiler to specify that constants may be accessed as read-write data (as in ANSI C). The compiler's default behavior assumes that data referenced through `const` pointers never changes.

The `-const-read-write` switch changes the compiler's behavior to match the ANSI C assumption, which is that other `non-const` pointers may be used to change the data at some point.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Cross-Core Blackfin C/C++ Compiler > Language Settings > Pointers to const may point to non-const data.*

-const-strings

The `-const-strings` (`const`-qualify strings) switch directs the compiler to mark string literals as `const`-qualified. This is the default mode.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Cross-Core Blackfin C/C++ Compiler > Language Settings > Literal strings are const.*

See also -no-const-strings.

-cplbs

The `-cplbs` (CPLBs are active) switch instructs the compiler to assume that all memory accesses will be validated by the Blackfin processor's memory protection hardware. This switch is best used in conjunction with the `-workaround` switch, as it allows the compiler to identify situations where the cacheability protection lookaside buffers (CPLBs) will avoid problems, thus avoiding the need for extra workaround instructions.

If only instruction CPLBs or data CPLBs are enabled, use -icplbs or -dcplbs, respectively.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor > CPLBs are enabled*.

See also -no-cplbs.

-D `macro`[=`definition`]

The −D (define macro) switch directs the compiler to define a macro. If you do not include the optional definition string, the compiler defines the macro as the string `1`. Note that the compiler processes −D switches on the command line before any −U (undefine macro) switches.

> **NOTE:** Add instances of this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > CrossCore Blackfin C/C++ Compiler > Preprocessor > Preprocessor definitions*.

-dcplbs

The −dcplbs (data CPLBs are active) switch instructs the compiler to assume that all data memory accesses will be validated by the Blackfin processor's memory protection hardware. This allows the compiler to identify situations where the cacheability protection lookaside buffers (CPLBs) will avoid problems the compiler would otherwise workaround (for example, anomaly 05-00-0428), improving code size and performance.

If both ICPLBs and DCPLBs are active, use -cplbs.

-decls-{weak|strong}

The −decls-weak and −decls-strong switches control how the compiler interprets uninitialized global variable definitions, such as int x;, when in C mode.

The −decls-strong switch treats this as equivalent to int x = 0;, specifying that other definitions of the same variable in other modules cause a "multiply-defined symbol" error. The −decls-weak switch treats this as equivalent to extern int x;, such as a declaration of a symbol that is defined in another module. The default is −decls-strong. ANSI C behavior is −decls-weak.

This switch has no effect when compiling in C++ mode.

> **NOTE:** Invoke this switch in the IDE by setting *Project > Properties > C/C++ Build > Settings > Tool Settings > CrossCore Blackfin C/C++ Compiler > Preprocessor > Treat uninitialized global vars as* to *zero-initialized*.

-dependency-add-target target

The −dependency-add-target switch adds *target* as another target that relies upon the dependencies in this build. Use this switch in conjunction with switches for emitting dependency information, such as -M.

For example, if you are building apple.doj from apple.c, the compiler's dependency output would indicate that apple.doj depends on apple.c. Using −dependency-add-target pear.doj causes the compiler to emit additional dependency information to indicate that apple.doj also depends on apple.c.

-double-size-{32 | 64}

The −double-size-32 (double is 32 bits) and −double-size-64 (double is 64 bits) switches specify the size of the double data type. The default is −double-size-32 (32-bit data type).

The -double-size-64 switch promotes double to a 64-bit data type, making it equivalent to long double. This switch does not affect the sizes of float or long double. Refer to Data Storage Formats for more information on data types.

> **NOTE:** Invoke this switch in the IDE by setting *Project > Properties > C/C++ Build > Settings > Tool Settings > CrossCore Blackfin C/C++ Compiler > Processor > Double size to the required value*.

-double-size-any

The -double-size-any switch specifies that the input source files make no use of double-typed data, and that resulting object files should be marked in such a way that will enable them to be linked against objects built with doubles, either 32 bits or 64 bits in size. Refer to Data Storage Formats for more information on data types.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Cross-Core Blackfin C/C++ Compiler > Processor > Allow mixing of sizes*.

-dry

The -dry (verbose dry run) switch directs the compiler to display main ccblkfn actions, but not to perform them.

-dryrun

The -dryrun (terse dry run) switch directs the compiler to display top-level ccblkfn actions, but not to perform them.

-E

The -E (stop after preprocessing) switch directs the compiler to stop after the C/C++ preprocessor runs (without compiling). The output (preprocessed source code) prints to the standard output stream unless the output file is specified with the -o *filename* switch.

-ED

The -ED (run after preprocessing to file) switch directs the compiler to write the output of the C/C++ preprocessor to a file named original_filename.i. After preprocessing, compilation proceeds normally.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Cross-Core Blackfin C/C++ Compiler > General > Generate preprocessed file*.

-EE

The -EE (run after preprocessing) switch directs the compiler to write the output of the C/C++ preprocessor to standard output. After preprocessing, compilation proceeds normally.

-eh

The -eh (enable exception handling) switch directs the compiler to allow C++ code that contains catch statements and throw exceptions and other features associated with ANSI/ISO standard C++ exceptions. When this switch is enabled, the compiler defines the macro __EXCEPTIONS as 1.

If used when compiling C programs, without the -c++ (C++ mode) switch, the -eh switch directs the compiler to generate exceptions tables but does not change the language accepted. In this case, __EXCEPTIONS is not defined.

The `-eh` switch also causes the compiler to define `__ADI_LIBEH__` during the linking stage so that appropriate sections can be activated in the `.ldf` file, and the program can be linked with a library built with exceptions enabled.

Object files created with exceptions enabled may be linked with objects created without exceptions enabled. However, exceptions can only be thrown from and caught, and cleanup code executed, in modules compiled with `-eh`. If an attempt is made to throw an exception through the execution of a function not compiled `-eh`, then `abort` or the function registered with `set_terminate` is called. See also Exceptions Tables Pragma.

In non-threaded applications, the buffer used for the passing of exception data is not returned to the heap on application exit. This is to avoid unnecessary code and will have no impact on behavior.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Cross-Core Blackfin C/C++ Compiler > Language Settings > C++ exceptions and RTTI.*

See also -no-eh.

-enum-is-int

The `-enum-is-int` switch ensures that the type of an `enum` is `int`. By default, the compiler may define enumeration types with integral types larger than `int`, if `int` is insufficient to represent all the values in the enumeration. This switch prevents the compiler from selecting a type wider than `int`. See Enumeration Type Implementation Details for more information.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Cross-Core Blackfin C/C++ Compiler > Language Settings > Enumerated types are always int.*

-expand-symbolic-links

The `-expand-symbolic-links` (expand symbolic links) switch directs the compiler to recognize Cygwin path extensions (see Cygwin Path Support) within command-line switches and `#include` preprocessor directives. This option is disabled by default.

See also -no-expand-symbolic-links.

-expand-windows-shortcuts

The `-expand-windows-shortcuts` (expand Windows shortcuts) switch directs the compiler to recognize Windows shortcuts (Windows Shortcut Support) within command-line switches and `#include` preprocessor directives. (This option is disabled by default.)

See also -no-expand-windows-shortcuts.

-extra-keywords

The `-extra-keywords` (enable short-form keywords) switch directs the compiler to recognize the Analog Devices keyword extensions to ANSI/ISO standard C/C++ without leading underscores, which can affect conforming ANSI/ISO C/C++ programs. This is the default mode.

When the `-extra-keywords` switch is in effect, the same set of keywords is available regardless of language mode; which of those keywords is an extension, and which is a part of the language Standard, varies according to the current language mode, as indicated in the *Extra Keywords Supported According to Language Mode* table.

Use the -no-extra-keywords switch to disallow support for the additional keywords. The following table provides a list and a brief description of keyword extensions.

Table 2-10: Extra Keywords Supported According to Language Mode

Language Mode	Extra Keywords Supported Beyond the Language Standard
C89 Mode	`inline`, `asm`, `bank`, `section`, `bool`, `true`, `false`, `restrict`, `segment`
C99 Mode	`asm`, `bank`, `section`, `bool`, `true`, `false`, `segment`
C++ Mode	`bank`, `section`, `restrict`, `segment`

-file-attr `name[=value]`

The `-file-attr` (file attribute) switch directs the compiler to add the specified attribute name/value pair to all the files it compiles. To add multiple attributes, use the switch multiple times. If `= value` is omitted, the default value of "1" is used. See File Attributes for more information about attributes, and what automatic attributes the compiler emits.

See also -auto-attrs and -no-auto-attrs.

> **NOTE:** Add instances of this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > CrossCore Blackfin C/C++ Compiler > General > Additional attributes*.

-fixed-point-io

The `-fixed-point-io` (use fixed-point I/O library) switch links the application with a variant of the Analog Devices I/O library with support for printing `fract` and `accum` types in decimal format with the printf family of functions using the `%k`, `%K`, `%r`, and `%R` conversion specifiers. This library provides output that adheres to the embedded C Technical Report 18037 at the expense of increased code size footprint. Linking with the default I/O library provides output using the `%k`, `%K`, `%r`, and `%R` specifiers only in hexadecimal format. Note that the Analog Devices libraries contain a faster implementation of C standard I/O than the alternative third-party I/O library (see -full-io) but that the functionality provided is not as comprehensive. For details, refer to stdio.h in the *C/C++ Run-Time Library* chapter.

This switch passes the `_ADI_FX_LIBIO` macro to the compiler and linker.

> **NOTE:** Invoke this switch in the IDE by setting *Project > Properties > C/C++ Build > Settings > Tool Settings > Linker > Processor > I/O libraries to High-performance I/O with support for fixed-point types*.

See also -full-io and -no-full-io.

-flags{-asm | -compiler | -ipa | -lib | -link | -mem | -prelink} switch[, switch2[,...]]

The `-flags` (command-line input) switch directs the compiler to pass command-line switches to the other build tools.

Versions of this switch are listed in the *Switches Passed to Other Build Tools* table.

Table 2-11: Switches Passed to Other Build Tools

Switch	Tool
`-flags-asm`	Assembler
`-flags-compiler`	Compiler executable
`-flags-ipa`	IPA Solver
`-flags-lib`	Library Builder (`elfar.exe`)
`-flags-link`	Linker
`-flags-mem`	Memory Initializer
`-flags-prelink`	Prelinker

-force-circbuf

The `-force-circbuf` (circular buffer) switch instructs the compiler to use circular buffer facilities, even if the compiler cannot verify that the circular index or pointer is always within the range of the buffer. Without this switch, the compiler's default behavior is conservative, and does not use circular buffers unless it can verify that the circular index or pointer is always within the circular buffer range. See Circular Buffer Built-In Functions.

> **NOTE:** Invoke this switch in the IDE by setting *Project > Properties > C/C++ Build > Settings > Tool Settings > CrossCore Blackfin C/C++ Compiler > Language Settings > Circular buffer generation* to *Even when pointer may be outside buffer range*.

-force-link

The `-force-link` (force stack frame creation) switch directs the compiler to create a new stack frame for leaf functions.

This is selected by default if the -g switch is selected as it improves the quality of debugging information, but can be switched off with `-no-force-link`. When -p is selected, this switch is always in force.

See also -no-force-link.

-fp-associative

The `-fp-associative` switch permits the compiler to treat floating-point addition and multiplication as associative operations, allowing it to reorder sequences of additions or multiplications where that is beneficial for performance. Due to different rounding of intermediate results, this may change the result of such sequences.

This switch is on by default if optimization is enabled.

See also -no-fp-associative.

-full-io

The `-full-io` switch links the application with a third-party, proprietary I/O library. The third-party I/O library provides a complete implementation of the ANSI C Standard I/O functionality at the cost of performance (compared to the Analog Devices I/O library). For details, see stdio.h in the *C/C++ Run-Time Library* chapter.

In addition, the third-party library prints fixed-point values in decimal format with the printf family of functions using the `%k`, `%K`, `%r`, and `%R` conversion specifiers.

This switch defines the `_DINKUM_IO` macro during compilation and linking.

> **NOTE:** Invoke this switch in the IDE by setting *Project > Properties > C/C++ Build > Settings > Tool Settings > Linker > Processor > I/O libraries to Full ANSI C compliant I/O*.

See also -no-full-io and -fixed-point-io.

> **NOTE:** Because certain operations in the full C++ Standard library rely on wide-character I/O support, use of the full Standard C++ library may require the full ANSI-compliant I/O libraries. Failure to use this library can result in the following linker error:
> ```
> [Error li1021] The following symbols referenced in processor 'p0' could
> not be resolved:
>
> 'wide_character_IO_not_supported_without_full_io' referenced from
> 'libio.dlb[widechar.doj]'
> ```
> To enable use of Full ANSI-compliant I/O library, use the -full-io compiler switch, or select *Full ANSI C compliant I/O (-full-io)* in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > CrossCore Blackfin Linker > Libraries> I/O Libraries*.

-full-version

The `-full-version` (display version) switch directs the compiler to display version information for all the compilation tools as they process each file.

-fx-contract

The `-fx-contract` switch sets the default state of `FX_CONTRACT` to `ON`, which is the default setting. This switch controls the performance and accuracy of arithmetic on the native fixed-point types `fract` and `accum`. See FX_CONTRACT for more information.

See also -no-fx-contract.

-fx-rounding-mode-biased

The `-fx-rounding-mode-biased` switch sets the default state of `FX_ROUNDING_MODE` to `BIASED`. This switch controls the rounding behavior of arithmetic on the native fixed-point types `fract` and `accum`. See Setting the Rounding Mode for more information. It should be used in conjunction with the `set_rnd_mod_biased()` built-in function, described in Changing the RND_MOD Bit.

-fx-rounding-mode-truncation

The `-fx-rounding-mode-truncation` switch sets the default state of `FX_ROUNDING_MODE` to `TRUNCATION`, which is the default setting. This switch controls the rounding behavior of arithmetic on the native fixed-point types `fract` and `accum`. See Setting the Rounding Mode for more information.

-fx-rounding-mode-unbiased

The `-fx-rounding-mode-unbiased` switch sets the default state of `FX_ROUNDING_MODE` to `UNBIASED`. This switch controls the rounding behavior of arithmetic on the native fixed-point types `fract` and `accum`. See Setting the Rounding Mode for more information. It should be used in conjunction with the `set_rnd_mod_unbiased()` built-in function, described in Changing the RND_MOD Bit.

-g

The `-g` (generate debugging information) switch directs the compiler to output symbols and other information used by the debugger.

If the `-g` switch is used with the `-O` (enable optimization) switch, the compiler performs standard optimizations. The compiler also outputs symbols and other information to provide limited source-level debugging. This combination of options provides line debugging and global variable debugging.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Cross-Core Blackfin C/C++ Compiler > General > Generate debug information*.

> **NOTE:** When the `-g` and `-O` switches are specified, no debug information is available for local variables and the standard optimizations can sometimes rearrange program code in a way that produces inaccurate line number information. For full debugging capabilities, use the `-g` switch without the `-O` switch.

-glite

The `-glite` (lightweight debugging) switch can be used on its own, or in conjunction with the `-g` compiler switch. When this switch is enabled, it instructs the compiler to remove any unnecessary debug information for the code that is compiled.

When used on its own, the switch also enables the `-g` option.

> **NOTE:** This switch can be used to reduce the size of object and executable files, but will have no effect on the size of the code loaded onto the target.

-gnu-style-dependencies

The `-gnu-style-dependencies` switch changes the format in which dependency information, such as that produced by the `-M` switch, is produced, so that it matches the format used by the GNU `make` program. The differences are shown in the *Effect of -gnu-style-dependencies Switch* table.

Table 2-12: Effect of -gnu-style-dependencies Switch

Format	Without-gnu-style-dependencies	With-gnu-style-dependencies
Quoting	Yes (`"foo"`)	No (`foo`)

Table 2-12: Effect of -gnu-style-dependencies Switch (Continued)

Format	Without-gnu-style-dependencies	With-gnu-style-dependencies
Whitespace	Quoted (`"x y"`)	Escaped with backslash (`x \ y`)
Directory separators	Backslash (`\`)	Forward slash (`/`)
Path form	Canonical (`"c:\foo\bar"`)	Relative (`../bar`)

The IDE applies this switch automatically.

-H

The `-H` (list headers) switch directs the compiler to output a list of the files included by the preprocessor via the `#include` directive, without compiling. The `-o filename` switch may be used to redirect the list to a file.

-HH

The `-HH` (list headers and compile) switch directs the compiler to print to the standard output file stream a list of the files included by the preprocessor via the `#include` directive. After preprocessing, compilation proceeds normally.

-h[elp]

The `-h` or `-help` (command-line help) switch directs the compiler to output a list of command-line switches with a brief syntax description.

-I *directory*`[{,|;}` *directory...*`]`

The `-I` (include search directory) switch directs the C/C++ preprocessor to append the directory (or directories) to the search path for `include` files. This option can be specified more than once; all specified directories are added to the search path.

> **NOTE:** Add instances of this switch in the IDE via ***Project > Properties > C/C++ Build > Settings > Tool Settings > CrossCore Blackfin C/C++ Compiler > Preprocessor > Additional include directories***.

Include files, whose names are not absolute path names and that are enclosed in "..." when included, are searched for in the following directories in this order:

1. The directory containing the current input file (the primary source file or the file containing the `#include`).

2. Any directories specified with the `-I` switch in the order they are listed on the command line.

3. Any directories on the standard list: `<install_path>\...\include`.

> **NOTE:** If a file is included using the <...> form, this file is only searched for by using directories defined in items 2 and 3 above.

Invoke this switch with the ***Additional include directories*** text field located in the CCES ***Tool Settings*** dialog box (***Compiler > Preprocessor*** page).

-I-

The `-I-` (start include directory list) switch establishes the point in the include directory list at which the search for header files enclosed in angle brackets begins. Normally, for header files enclosed in double quotes, the compiler searches in the directory containing the current input file; then the compiler reverts back to looking in the directories specified with the `-I` switch; and then the compiler searches in the standard `include` directory.

It is possible to replace the initial search (within the directory containing the current input file) by placing the `-I-` switch at the point on the command line where the search for all types of header file begins. All `include` directories on the command line specified before the `-I-` switch are used only in the search for header files that are enclosed in double quotes.

> **NOTE:** This switch removes the directory containing the current input file from the `include` directory list.

-i

The `-i` (less includes) switch may be used with the `-H`, `-HH`, `-M`, or `-MM` switches to direct the compiler to only output header details (`-H`, `-HH`) or makefile dependencies (`-M`, `-MM`) for `include` files specified in double quotes.

-icplbs

The `-icplbs` (instruction CPLBs are active) switch instructs the compiler to assume that all instruction memory accesses will be validated by the Blackfin processor's memory protection hardware. This allows the compiler to identify situations where the cacheability protection lookaside buffers (CPLBs) will avoid problems the compiler would otherwise workaround (for example, anomaly 05-00-0426), improving code size and performance.

If both ICPLBs and DCPLBs are active, use the -cplbs switch.

-include *filename*

The `-include` *filename* (include file) switch directs the preprocessor to process the specified file before processing the regular input file. Any `-D` and `-U` options on the command line are processed before an `-include` file.

-ipa

The `-ipa` (interprocedural analysis) switch turns on interprocedural analysis (IPA) in the compiler. This option enables optimization across the entire program, including between source files that were compiled separately. If used, the `-ipa` switch should be applied to all C and C++ files in the program. For more information, see Interprocedural Analysis. Specifying `-ipa` also implies setting the -O[0|1] switch.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > General > Interprocedural optimization*.

-jcs2l

The `-jcs2l` switch requests the linker to convert compiler-generated `short` jumps and calls to `long` jumps and calls when necessary.

Blackfin+ processors have jump and call instructions with offset fields large enough to reach the entire address range, but previous Blackfin processors do not. When the range of the available instructions is insufficient, the target address is loaded into the P1 register and an indirect jump or call to P1 is issued.

This switch is enabled by default. See also -no-jcs2l.

-L *directory[{,|;} directory]*

The -L *directory* (library search directory) switch directs the linker to append the directory (or directories) to the search path for library files.

> **NOTE:** Add instances of this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Linker > General > Search directories*.

-l *library*

The -l *library* (link library) switch directs the linker to search the library for functions and global variables when linking. The library name is the portion of the file name between the lib prefix and the .dlb extension. For example, the -lc compiler switch directs the linker to search in the library named c. This library resides in a file named libc.dlb.

List all object files on the command line before listing libraries using the -l switch. When a reference to a symbol is made, the symbol definition will be taken from the left-most object or library on the command line that contains the global definition of that symbol. If two objects on the command line contain definitions of the symbol x, x will be taken from the left-most object on the command line that contains a global definition of x.

If one of the definitions for x comes from user objects, and the other comes from a user library, and the library definition should be overridden by the user object definition, it is important that the user object comes before the library on the command line.

Libraries included in the default .ldf file are searched last for symbol definitions.

> **NOTE:** Add instances of this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Linker > General > Additional libraries and object files*.

-list-workarounds

The -list-workarounds (list supported errata workarounds) switch displays a list of all errata workarounds which the compiler supports. See Controlling Silicon Revision and Anomaly Workarounds Within the Compiler for details of valid workarounds and the interaction of the -si-revision *version* , -workaround *workaround_id[, workaround_id]* , and -no-workaround *workaround_id[, workaround_id]* switches.

-M

The -M (generate make rules only) switch directs the compiler not to compile the source file, but to output a rule suitable for the make , describing the dependencies of the main program file.

The format of the make rule output by the preprocessor is:

```
object-file: include-file ...
```

-MD

The -MD (generate make rules and compile) switch directs the preprocessor to print to a file called original_filename.d a rule describing the dependencies of the main program file. After preprocessing, compilation proceeds normally.

See also -Mo `filename` .

-MM

The `-MM` (generate make rules and compile) switch directs the preprocessor to print to the standard output stream a rule describing the dependencies of the main program file. After preprocessing, compilation proceeds normally.

-Mo `filename`

The `-Mo filename` (preprocessor output file) switch directs the compiler to use `filename` for the output of `-MD` or `-ED` switches.

-Mt `name`

The `-Mt name` (output make rule for the named source) switch modifies the target of generated dependencies, renaming the target to name. This switch is in effect only when used in conjunction with the `-M` or `-MM` switch.

-map `filename`

The `-map filename` (generate a memory map) switch directs the linker to output a memory map of all symbols. The map file name corresponds to the `filename` argument. For example, if the file name argument is `test`, the map file name is `test.xml`. The `.xml` extension is added where necessary.

-mem

The `-mem` (invoke memory initializer) switch causes the compiler to invoke the Memory Initializer after linking the executable file. The Memory Initializer can be controlled through the -flags{-asm | -compiler | -ipa | -lib | -link | -mem | -prelink} switch[, switch2[,...]] switch or disabled using the -no-mem switch.

For more information, see:

- *StartUp Code* in the *System Run-Time Documentation*

- *Memory Initializer* in the *Linker and Utilities Manual*

-multiline

The `-multiline` switch directs the compiler to allow string literals to span multiple lines without the need for a backslash character "\" at the end of each line. This is the default mode.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Language Settings > Allow multi-line character strings*.

See also -no-multiline.

-never-inline

The `-never-inline` switch instructs the compiler to ignore the `inline` qualifier on function definitions, so that no calls to such functions will be inlined.

> **NOTE:** Invoke this switch in the IDE by setting *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > General > Inlining to Never*.

See also -always-inline.

-no-alttok

The `-no-alttok` (disable alternative tokens) switch directs the compiler not to accept digraph sequences in the source files. This switch is enabled by default in C++ mode, and disabled by default in C89 and C99 modes. In C++ mode, the switch also controls the acceptance of alternative operator keywords.

See also -alttok .

-no-annotate

The `-no-annotate` (disable assembly annotations) switch directs the compiler not to annotate assembly files generated by the compiler. By default, whenever optimizations are enabled, all assembly files generated by the compiler are annotated with information on the performance of the generated assembly. See Assembly Optimizer Annotations in the *Optimal Performance from C/C++ Source Code* chapter for details on this feature.

> **NOTE:** Invoke this switch in the IDE by clearing *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > General > Generate annotations*.

See also -annotate.

-no-annotate-loop-instr

The `-no-annotate-loop-instr` switch disables the production of additional loop annotation information by the compiler. This is the default mode.

See also -annotate-loop-instr.

-no-assume-vols-are-mmrs

When the compiler has to apply workarounds for silicon errata, it takes a conservative approach concerning `volatile`-qualified accesses to arbitrary memory. By default, the compiler assumes that such memory accesses may be to memory-mapped registers (MMRs), and therefore must be protected against any errata that affect MMR accesses.

The `-no-assume-vols-are-mmrs` switch disables this assumption, so that arbitrary `volatile`-qualified memory will not be considered affected by MMR-related errata. Specific MMR accesses, such as via a literal pointer or the Memory-Mapped Register Access Built-In Functions, will still receive such workarounds. For more information, see Controlling Silicon Revision and Anomaly Workarounds Within the Compiler.

-no-auto-attrs

The `-no-auto-attrs` (no automatic attributes) switch directs the compiler not to emit automatic attributes based on the files it compiles. Emission of automatic attributes is enabled by default. See File Attributes for more information about attributes, and what automatic attributes the compiler emits.

> **NOTE:** Invoke this switch in the IDE by clearing *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > General > Auto-generated attributes*.

See also -auto-attrs and -file-attr *name[=value]* .

-no-bss

The `-no-bss` switch causes the compiler to keep both zero-initialized and non-zero-initialized data in the same data section, rather than separating zero-initialized data into a different, BSS-style section.

See also -bss.

-no-circbuf

The `-no-circbuf` (no circular buffer) switch directs the compiler not to automatically use circular buffer mechanisms (such as for referencing `array[i % n]`). The use of the `circindex()` and `circptr()` functions (that is, explicit circular buffer operations) is not affected.

> **NOTE:** Invoke this switch in the IDE by setting *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Language Settings > Circular Buffer Generation to Never*.

-no-const-strings

The `-no-const-strings` switch directs the compiler not to make string literals `const` qualified.

> **NOTE:** Invoke this switch in the IDE by clearing *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Language Settings > Literal strings are const*.

See also -const-strings.

-no-cplbs

The `-no-cplbs` (CPLBs are not enabled) switch informs the compiler that neither Data CPLBs nor Instruction CPLBs are enabled, and therefore the compiler should be conservative when generating code that will cause speculative accesses to memory. This is the default.

See also -cplbs, -dcplbs, and -icplbs.

-no-defs

The `-no-defs` (disable defaults) switch directs the compiler not to define any default preprocessor macros, include directories, library directories or libraries.

-no-eh

The `-no-eh` (disable exception handling) switch directs the compiler to disallow ANSI/ISO C++ exception handling. This is the default mode.

See also -eh.

-no-expand-symbolic-links

The `-no-expand-symbolic-links` switch directs the compiler not to recognize Cygwin path entities (see Cygwin Path Support) within command-line paths and preprocessor `#include` directives. This option is enabled by default.

See also -expand-symbolic-links.

-no-expand-windows-shortcuts

The `-no-expand-windows-shortcuts` switch directs the compiler not to recognize Windows shortcut entities (see Windows Shortcut Support) within command-line paths and preprocessor `#include` directives. This option is enabled by default.

See also -expand-windows-shortcuts.

-no-extra-keywords

The `-no-extra-keywords` (disable short-form keywords) switch directs the compiler not to recognize Analog Devices keyword extensions that might conflict with valid C/C++ identifiers, for example, `section`. Alternate keywords (prefixed with two leading underscores, such as `__section`) continue to work.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Language Settings > Disable Analog Devices extension keywords*.

See also -extra-keywords.

-no-force-link

The `-no-force-link` (do not force stack frame creation) switch directs the compiler not to create a new stack frame for leaf functions.

This switch is most useful in combination with the -g switch when debugging optimized code. When optimization is requested, the compiler does not generate stack frames for functions that do not need them; this improves the size and speed of the code, but reduces the quality of information displayed in the debugger. Therefore, when the `-g` switch is used, the compiler by default always generates a stack frame. Consequently, the code generated with the `-g` switch differs from the code generated without using this switch and may result in different behavior. The `-no-force-link` switch causes the same code to be generated regardless of whether `-g` is used.

See also -force-link.

-no-fp-associative

By default, when optimization is enabled, the compiler may treat floating-point addition and multiplication as associative operations, to allow sequences of additions or multiplications to be reordered where that is beneficial for performance. Due to different rounding of intermediate results, this may change the result of such sequences.

The `-no-fp-associative` switch disables this behavior.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Language Settings > Do not treat floating-point operations as associative*.

See also -fp-associative.

-no-full-io

The `-no-full-io` switch links the application with the Analog Devices I/O library, which contains a faster implementation of C Standard I/O than the alternative third-party I/O library; see -full-io. The functionality provided by the Analog Devices I/O library is not as comprehensive as the third-party I/O library. For details, refer to stdio.h in the *C/C++ Run-Time Library* chapter.

This switch defines the `_ADI_LIBIO` macro during compilation and linking. This switch is enabled by default.

-no-fx-contract

The `-no-fx-contract` switch sets the default state of `FX_CONTRACT` to `OFF`. This switch controls the performance and accuracy of arithmetic on the native fixed-point types `fract` and `accum`. See FX_CONTRACT for more information.

See also -fx-contract.

-no-int-to-fract

The `-no-int-to-fract` (disable conversion of integer to fractional arithmetic) switch directs the compiler not to turn integer arithmetic into fractional arithmetic.

For example, the following statement may be changed, by default, into a fractional multiplication.

```
short a = ((b*c)>>15);
```

The saturation properties of integer and fractional arithmetic are different; therefore, if the expression overflows, the results differ. Specifying the `-no-int-to-fract` switch disables this optimization. Note that the switch does not affect arithmetic on the native fixed-point types `fract` and `accum`.

-no-jcs2l

The `-no-jcs2l` switch prevents the linker from converting compiler-generated `short` jumps to `long` jumps or indirect jumps via register `P1`.

> **NOTE:** The BF7xx family of processors support an absolute addressed call instruction, so expansion is performed without using P1, and the `-no-jcs2l` switch has no effect.

See also -jcs2l.

-no-mem

The `-no-mem` (disable memory initialization) switch causes the compiler not to invoke the Memory Initializer after linking the executable. This is the default setting.

See also -mem.

-no-multiline

The `-no-multiline` switch directs the compiler to disallow string literals that span multiple lines without a `"\"` at the end of each line.

> **NOTE:** Invoke this switch by clearing *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Language Settings > Allow multi-line character strings*.

See also -multiline.

-no-progress-rep-timeout

The `-no-progress-rep-timeout` (disable progress message for long compilations) switch disables the diagnostic message issued by the compiler to indicate that it is still working when a function's compilation is taking an excessively long time. The message is disabled by default.

See also -progress-rep-timeout and -progress-rep-timeout-secs *secs* .

-no-rtcheck

The `-no-rtcheck` (disable run-time checking) switch directs the compiler to disable generation of additional code to check at runtime for common programming errors. This switch is the default, and is equivalent to specifying all of the following switches:

- -no-rtcheck-arr-bnd
- -no-rtcheck-div-zero
- -no-rtcheck-heap
- -no-rtcheck-null-ptr
- -no-rtcheck-shift-check
- -no-rtcheck-stack
- -no-rtcheck-unassigned

See also -rtcheck.

-no-rtcheck-arr-bnd

The `-no-rtcheck-arr-bnd` (do not check array bounds at runtime) switch directs the compiler not to generate additional code to verify that array accesses are within the bounds of the array.

> **NOTE:** Invoke this behavior in the IDE via the run-time checking options under *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor*.

See also -rtcheck.

-no-rtcheck-div-zero

The `-no-rtcheck-div-zero` (do not check for division by zero at runtime) switch directs the compiler not to generate additional code to verify that divisors are non-zero before performing division operations.

> **NOTE:** Invoke this behavior in the IDE via the run-time checking options under *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor*.

See also -rtcheck.

-no-rtcheck-heap

The `-no-rtcheck-heap` (do not check heap operations at runtime) switch directs the compiler not to link against the debugging version of the heap library.

NOTE: Invoke this behavior in the IDE via the run-time checking options under *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor*.

See also -rtcheck.

-no-rtcheck-null-ptr

The `-no-rtcheck-null-ptr` (do not check for NULL pointers at runtime) switch directs the compiler not to generate additional code to verify that pointers are not NULL, before dereferencing them.

NOTE: Invoke this behavior in the IDE via the run-time checking options under *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor*.

See also -rtcheck.

-no-rtcheck-shift-check

The `-no-rtcheck-shift-check` (do not check shift values at runtime) switch directs the compiler not to generate additional code to verify that, when shifting a value X by some amount Y, Y is a positive amount, and less than the number of bits used to represent X's type.

NOTE: Invoke this behavior in the IDE via the run-time checking options under *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor*.

See also -rtcheck.

-no-rtcheck-stack

The `-no-rtcheck-stack` (do not check for stack overflow at runtime) switch directs the compiler not to generate additional code to verify that increases in stack usage do not exceed the bounds of the available stack.

NOTE: Invoke this behavior in the IDE via the run-time checking options under *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor*.

See also -rtcheck.

-no-rtcheck-unassigned

The `-no-rtcheck-unassigned` (do not check variables are assigned at runtime) switch directs the compiler not to generate additional code to verify that variables have been assigned a value before they are used.

NOTE: Invoke this behavior in the IDE via the run-time checking options under *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor*.

See also -rtcheck.

-no-sat-associative

The `-no-sat-associative` (saturating addition is not associative) switch instructs the compiler not to consider saturating addition operations as associative: `(a+b)+c` may not be rewritten as `a+(b+c)`, when the addition operator saturates. The default is that saturating addition is not associative.

See also -sat-associative.

-no-saturation

The `-no-saturation` switch directs the compiler not to introduce faster operations in cases where the faster operation would saturate (if the expression overflowed) when the original operation would have wrapped the result. Note that since accumulator registers `A0` and `A1` will saturate if an accumulation overflows 40 bits, the `-no-saturation` switch will also prevent use of these registers for integer arithmetic when the compiler cannot be sure that saturation will not occur. The code produced may be less efficient than when the switch is not used.

Saturation is enabled by default when optimizing, and may be disabled by this switch. Saturation is disabled when not optimizing (this switch is the default when not optimizing).

Note that this switch does not affect the behavior of arithmetic with defined saturating semantics. For example, code written using the native fixed-point types `fract` and `accum`, or code written using fractional or accumulator built-in functions, will not change its behavior when this switch is used.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor > Do not introduce saturation to integer arithmetic*.

-no-std-ass

The `-no-std-ass` (disable standard assertions) switch prevents the compiler from defining the standard assertions.

See *-A name (tokens)* for the list of standard assertions.

-no-std-def

The `-no-std-def` (disable standard macro definitions) switch prevents the compiler from defining default preprocessor macro definitions.

-no-std-inc

The `-no-std-inc` (disable standard include search) switch directs the C/C++ preprocessor to search only for header files in the current directory and directories specified with the `-I` switch.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Preprocessor > Ignore standard include paths*.

-no-std-lib

The `-no-std-lib` (disable standard library search) switch directs the linker to limit its search for libraries to directories specified with the *-L directory[{,|;} directory]* switch. The compiler also defines `__NO_STD_LIB` during the linking stage and passes it to the linker, so that the `SEARCH_DIR` directives in the `.ldf` file can be disabled.

-no-threads

The `-no-threads` (disable thread-safe build) switch directs the compiler to link against the non-thread-safe variants of the C/C++ variants of the run-time libraries. This is the default.

See also *-threads*.

-no-utility-rom

The `-no-utility-rom` (do not use Utility ROM) switch directs the tools not to link against the library functions in the processor's ROM, when building the executable image. This switch passes a definition of the macro `NO_UTILITY_ROM` to the linker.

This switch is only supported for the ADSP-BF592-A and ADSP-BF7xx family processors. It is disabled by default when building for silicon revisions that can use the ROM.

See also -utility-rom.

-no-workaround *workaround_id[, workaround_id]*

The `-no-workaround` *workaround_id* switch (disable avoidance of specific errata) switch disables compiler code generator workarounds for specific hardware errata. See Controlling Silicon Revision and Anomaly Workarounds Within the Compiler for details of valid workarounds and the interactions of the `-si-revision`, `-workaround`, and `-no-workaround` switches.

See also -workaround *workaround_id[, workaround_id]* .

-no-zero-loop-counters

The `-no-zero-loop-counters` switch directs the compiler to not zero loop counter registers on function exit. This is the default mode.

Use the -zero-loop-counters switch to enable the zeroing of loop counter registers on function exit.

-O[0|1]

The `-O` (enable optimizations) switch directs the compiler to produce code that is optimized for performance. Optimizations are not enabled by default for the compiler. (Note that the switch settings begin with the uppercase letter "O" and end with a digit-a zero or a one.) The `-O` or `-O1` switch turns on optimization, and `-O0` turns off all optimizations.

> **NOTE:** Invoke this switch in the IDE via ***Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > General > Enable optimization***.

-Oa

The `-Oa` (automatic function inlining) switch enables the inline expansion of C/C++ functions, which are not necessarily declared inline in the source code. The amount of auto-inlining the compiler performs is controlled using the -Ov *num* (optimize for speed versus size) switch. Therefore, the use of `-Ov100` indicates that as many functions as possible will be auto-inlined, whereas `-Ov0` prevents any function from being auto-inlined. Specifying `-Oa` implies the use of `-O`.

> **NOTE:** Invoke this switch in the IDE by setting ***Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > General > Inlining to Automatic***.

-Os

The $-Os$ (enable code size optimization) switch directs the compiler to produce code that is optimized for size. This is achieved by performing all optimizations except those that increase code size. The optimizations not performed include loop unrolling and jump avoidance.

-Ov *num*

The $-Ov$ *num* (optimize for speed versus size) switch informs the compiler of the relative importance of speed versus size, when considering whether such trade-offs are worthwhile. The *num* variable should be an integer between 0 (purely size) and 100 (purely speed).

For any given optimization, the compiler modifies the code being generated. Some optimizations produce code that will execute in fewer cycles, but will require more code space. In such cases, there is a trade-off between speed and space.

The *num* variable indicates a sliding scale between 0 and 100, which is the probability that a linear piece of generated code (a "basic block") will be optimized for speed or for space. $-Ov0$ optimizes all blocks for space (equivalent to $-Os$), and $-Ov100$ optimizes all blocks for speed (equivalent to $-O$). At any point in between, the decision is based upon *num* and how many times the block is expected to be executed (the "execution count" of the block). The *-Ov Switch Optimization Curve* figure demonstrates this relationship.

For any given optimization where speed and size conflict, the potential benefit is dependent on the execution count. An optimization that increases performance at the expense of code size is considerably more beneficial if applied to the core loop of a critical algorithm than if applied to one-time initialization code or to rarely-used error-handling functions. If code only appears to be executed once, it will be optimized for space. As its execution count increases, so too does the likelihood that the compiler will consider the code increase worthwhile for the corresponding benefit in performance.

As *-Ov Switch Optimization Curve* figure shows, the $-Ov$ switch affects the point at which a given execution count is considered sufficient to switch optimization from "for space" to "for speed". Where *num* is a low value, the compiler is biased towards space, so a block's execution count has to be relatively high for the compiler to apply code-increasing transformations. Where *num* has a high value, the compiler is biased towards speed, so the same transformation will be considered valid for a much lower execution count.
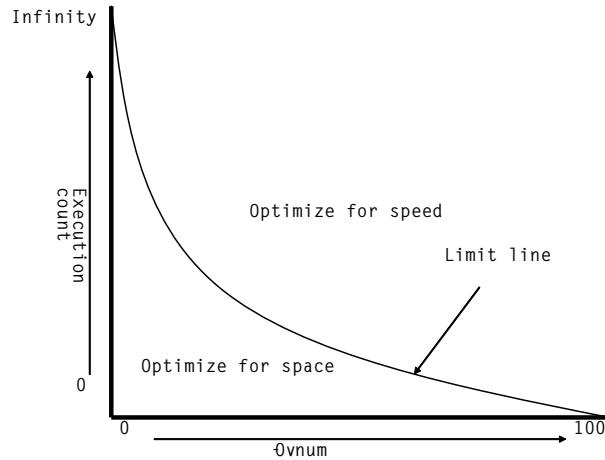
Figure 2-1: -Ov Switch Optimization Curve

The –Ov switch is most effective when used in conjunction with profile-guided optimization (PGO), where accurate execution counts are available. Without profile-guided optimization (see Optimization Control), the compiler makes estimates of the relative execution counts using heuristics.

> **NOTE:** Invoke this switch in the IDE by entering an appropriate value into *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > General > Optimize for code size/speed.*

For more information, see Using PGO in Function Profiling in the *Optimal Performance from C/C++ Source Code* chapter.

-O *filename*

The –o *filename* (output file) switch directs the compiler to use *filename* for the name of the final output file.

-overlay

The –overlay (program may use overlays) switch disables the propagation of register information between functions and forces the compiler to assume that all functions clobber all scratch registers. Note that this switch affects all functions in the source file and may result in a performance degradation. For information on disabling the propagation of register information only for specific functions, see #pragma overlay.

-overlay-clobbers *clobbered-regs*

The –overlay-clobbers (registers clobbered by overlay manager) switch identifies the set of registers clobbered by an overlay manager, if one is used. The compiler will assume that any call to an overlay-managed function will clobber the values in *clobbered-regs*, in addition to those clobbered by the function in question. A function is considered to be an overlay-managed function if the -overlay switch is specified, or if the function is marked with #pragma overlay.

The *clobbered-regs* is a single string formatted as per the argument to #pragma regs_clobbered, except that individual components of the list may also be separated by commas.

NOTE: Whitespace and semicolons are valid separators for the components of the list, but must be properly quoted when being passed to the compiler.

Examples:

```
ccblkfn -O t.c -overlay -overlay-clobbers r0,r1
ccblkfn -O t.c -overlay -overlay-clobbers Dscratch
ccblkfn -O t.c -overlay -overlay-clobbers "p0 p1;r0"
```

-P

The -P (omit line numbers) switch directs the compiler to stop after the C/C++ preprocessor runs (without compiling) and to omit #line preprocessor directives (with line number information) in the output from the preprocessor. The -C switch can be used with the -P switch to retain comments.

-PP

The -PP (omit line numbers and compile) switch is similar to the -P switch; however, it does not halt compilation after preprocessing.

-p

The -p (generate instrumented profiling) switch directs the compiler to generate the additional instructions needed to profile the program by recording the number of cycles spent in each function.

The -p switch writes the data to a .prf file. For more information on profiling, see the Profiling With Instrumented Code chapter.

NOTE: Instrumented profiling generates calls to supporting libraries to implement this functionality. This increases the stack space used by your application.

NOTE: Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Profiling > Enable compiler instrumented profiling*.

-path {-asm | -compiler | -ipa | -lib | -link | -prelink} *pathname*

The -path-{asm|compiler|ipa|lib|link|prelink} *pathname* (tool location) switch directs the compiler to use the specified component in place of the default-installed version of the compilation tool. The component comprises a relative or absolute path to its location. Respectively, the tools are the assembler, compiler, IPA solver, library builder, linker and prelinker. Use this switch when overriding the normal version of one or more of the tools. The -path-{...} switch also overrides the directory specified by the -path-install *directory* switch.

-path-install *directory*

The -path-install *directory* (installation location) switch directs the compiler to use the specified directory as the location for all compilation tools instead of the default path. This is useful when working with multiple versions of the tool set.

NOTE: You can selectively override this switch with the -path-{asm|compiler| lib|link} switch.

-path-output *directory*

The -path-output *directory* (non-temporary files location) switch directs the compiler to place output files in the specified directory.

-path-temp *directory*

The -path-temp *directory* (temporary files location) switch directs the compiler to place temporary files in the specified directory.

-pgo-session *session-id*

The -pgo-session *session-id* (specify PGO session identifier) switch is used with profile-guided optimization. It has the following effects:

- When used with the -pguide switch, the compiler associates all counters for this module with the session identifier *session-id*.

- When used with a previously-gathered profile (.pgo file), the compiler ignores the profile contents, unless they have the same *session-id* identifier.

This is most useful when the same source file is being built in more than one way (for example, different macro definitions, or for multiprocessors) in the same application. Each variant of the build can have a different *session-id* associated with it, which means that the compiler will be able to identify which parts of the gathered profile are to be used when optimizing for the final build.

If each source file is built only in a single manner within the system (the usual case), the -pgo-session switch is not needed.

> **NOTE:** Invoke this switch in the IDE by entering a suitable name into the *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Profile-guided Optimization > PGO Session name field*.

For more information, see Using PGO in Function Profiling in the Achieving Optimal Performance From C/C++ Source Code chapter.

-pguide

The -pguide (PGO) switch causes the compiler to add instrumentation to gather a profile (a .pgo file) as the first stage of performing profile-guided optimization.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Profile-guided Optimization > Prepare application to create new profile*.

For more information, see Using PGO in Function Profiling in the *Optimal Performance from C/C++ Source Code* chapter.

-pplist *filename*

The -pplist *filename* (preprocessor listing) switch directs the preprocessor to output a listing to the named file. When more than one source file is preprocessed, the listing file contains information about the last file processed. The generated file contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the compiler.

Key characters are described in the *Key Characters* table.

Table 2-13: Key Characters

Character	*Description*
N	Normal line of source
X	Expanded line of source
S	Line of source skipped by `#if` or `#ifdef`
L	Change in source position
R	Diagnostic message (remark)
W	Diagnostic message (warning)
E	Diagnostic message (error)
C	Diagnostic message (catastrophic error)

-proc *processor*

The `-proc` *processor* (target processor) switch directs the compiler to produce code suitable for the specified processor. Refer to the CCES online help for the list of supported Blackfin processors.

For example:

```
ccblkfn -proc ADSP-BF533 -o bin/p1.doj p1.asm
```

> **NOTE:** If no target is specified with the `-proc` switch, the default processor is set to `ADSP-BF532`.

When compiling with the `-proc` switch, the appropriate processor macro is defined as "1". The compiler additionally defines the `__ADSPBLACKFIN__` and `__ADSPLPBLACKFIN__` preprocessor macros as "1".

For example, when `-proc ADSP-BF531` is used, the compiler predefines the `__ADSPBF531__`, `__ADSPBLACKFIN__`, and `__ADSPLPBLACKFIN__` macros to "1".

> **NOTE:** See also -si-revision *version* for more information on the silicon revision of the specified processor.

-prof-hw

The `-prof-hw` switch instructs the compiler to generate profiling code that shall be run on hardware (rather than on the simulator). The switch requires a supported profiling switch to also be specified on the command line. Supported profiling method is Profile-guided Optimization (-pguide).

> **NOTE:** Instrumented profiling (-p) does not differentiate between execution on hardware or simulator, and can be executed on both targets. It does not require the `-prof-hw` switch.

> **NOTE:** Profiling on hardware may rely on code instrumentation and may make calls into supporting libraries to implement this functionality. Be aware that this can considerably increase the necessary stack space for your application.

NOTE: Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Profile-guided Optimization > Gather profile using hardware.*

-progress-rep-func

The `-progress-rep-func` switch provides feedback on the compiler's progress that may be useful when compiling and optimizing very large source files. It issues a warning message each time the compiler starts compiling a new function. The warning message is a remark that is disabled by default, and this switch enables the remark as a warning. The switch is equivalent to `-Wwarn=cc1472`.

-progress-rep-opt

The `-progress-rep-opt` switch provides feedback on the compiler's progress that may be useful when compiling and optimizing a very large, complex function. It issues a warning message each time the compiler starts a new optimization pass on the current function. The warning message is a remark that is disabled by default, and this switch enables the remark as a warning. The switch is equivalent to `-Wwarn=cc1473`.

-progress-rep-timeout

The `-progress-rep-timeout` switch issues a diagnostic message if the compiler exceeds a time limit during compilation. This indicates the compiler is still operating, but is taking a long time.

See also -no-progress-rep-timeout.

-progress-rep-timeout-secs *secs*

The `-progress-rep-timeout-secs` *secs* switch specifies how many seconds must elapse during a compilation before the compiler issues a diagnostic message about the length of time the compilation has used so far.

See also -no-progress-rep-timeout.

-R *directory[, directory]*

The `-R` *directory* (add source directory) switch directs the compiler to add the specified directory to the list of directories searched for source files. Multiple source directories can be presented as a comma-separated list.

The compiler searches for the source files in the order specified on the command line. The compiler searches the specified directories before reverting to the current directory. This switch is dependent on its position on the command line; that is, it effects only source files that follow it.

NOTE: Source files, whose file names begin with `/`, `./`, or `../`, (or Windows equivalent) or contain drive specifiers (on Windows platforms), are not affected by this option.

-R-

The `-R-` (disable source path) switch removes all directories from the standard search path for source files, effectively disabling this feature.

NOTE: This option is position-dependent on the command line; it only affects files following it.

-reserve *register[, register]*

The `-reserve` *register* (reserve register) switch directs the compiler not to use the specified registers. Only the `m3` register can be reserved.

-rtcheck

The `-rtcheck` (run-time checking) switch directs the compiler to generate additional code to check at runtime for common programming errors. This switch is equivalent to specifying all of the following switches:

- -rtcheck-arr-bnd
- -rtcheck-div-zero
- -rtcheck-heap
- -rtcheck-null-ptr
- -rtcheck-shift-check
- -rtcheck-stack
- -rtcheck-unassigned

> **NOTE:** Because of the additional overhead imposed by the checking code, this switch should only be employed during application development, and should not be used to build products for release.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor > Enable run-time checking*.

-rtcheck-arr-bnd

The `-rtcheck-arr-bnd` (check array bounds at runtime) switch directs the compiler to generate additional code to verify that array accesses are within the bounds of the array.

> **NOTE:** Because of the additional overhead imposed by the checking code, this switch should only be employed during application development, and should not be used to build products for release.

> **NOTE:** Invoke this switch in the IDE via the run-time checking options under *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor*.

See also -rtcheck.

-rtcheck-div-zero

The `-rtcheck-div-zero` (check for division by zero at runtime) switch directs the compiler to generate additional code to verify that divisors are non-zero before performing division operations.

> **NOTE:** Because of the additional overhead imposed by the checking code, this switch should only be employed during application development, and should not be used to build products for release.

> **NOTE:** Invoke this switch in the IDE via the run-time checking options under *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor*.

See also -rtcheck.

-rtcheck-heap

The `-rtcheck-heap` (check heap operations at runtime) switch directs the compiler to link against the debugging version of the heap library. For more information, see Heap Debugging in the ***Achieving Optimal Performance From C/C++ Source Code*** chapter.

> **NOTE:** Because of the additional overhead imposed by the checking code, this switch should only be employed during application development, and should not be used to build products for release.

> **NOTE:** Invoke this switch in the IDE via the run-time checking options under ***Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor***.

See also -rtcheck.

-rtcheck-null-ptr

The `-rtcheck-null-ptr` (check for NULL pointers at runtime) switch directs the compiler to generate additional code to verify that pointers are not NULL, before dereferencing them.

> **NOTE:** Because of the additional overhead imposed by the checking code, this switch should only be employed during application development, and should not be used to build products for release.

> **NOTE:** Invoke this switch in the IDE via the run-time checking options under ***Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor***.

See also -rtcheck.

-rtcheck-shift-check

The `-rtcheck-shift-check` (check shift values at runtime) switch directs the compiler to generate additional code to verify that, when shifting a value X by some amount Y, Y is a positive amount, and less than the number of bits used to represent X's type.

> **NOTE:** Because of the additional overhead imposed by the checking code, this switch should only be employed during application development, and should not be used to build products for release.

> **NOTE:** Invoke this switch in the IDE via the run-time checking options under ***Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor***.

See also -rtcheck.

-rtcheck-stack

The `-rtcheck-stack` (check for stack overflow at runtime) switch directs the compiler to generate additional code to verify, when increasing the amount of stack space in use, that the current stack bounds are not exceeded. For more information, see Stack Overflow Detection in the ***Optimal Performance from C/C++ Source Code*** chapter.

> **NOTE:** Because of the additional overhead imposed by the checking code, this switch should only be employed during application development, and should not be used to build products for release.

> **NOTE:** Invoke this switch in the IDE via the run-time checking options under *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor*.

See also -rtcheck.

-rtcheck-unassigned

The `-rtcheck-unassigned` (check variables are assigned at runtime) switch directs the compiler to generate additional code to verify that variables have been assigned a value before they are used.

> **NOTE:** Because of the additional overhead imposed by the checking code, this switch should only be employed during application development, and should not be used to build products for release.

> **NOTE:** Invoke this switch in the IDE via the run-time checking options under *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor*.

See also -rtcheck.

-S

The `-S` (stop after compilation) switch directs the compiler to stop compilation before running the assembler. The compiler outputs an assembly file with an `.s` extension.

-s

The `-s` (strip debug information) switch directs the compiler to remove debug information (symbol table and other items) from the output executable file during linking.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Linker > General > Strip all symbols*.

> **NOTE:** Executables produced by this switch are not suitable for use with the Memory Initializer. (See -mem for more information.)

-sat-associative

The `-sat-associative` (saturating addition is associative) switch instructs the compiler to consider saturating addition operations as associative; `(a+b)+c` may be rewritten as `a+(b+c)`, when the addition operator saturates. The default is that saturating addition is not associative.

See also -no-sat-associative.

-save-temps

The `-save-temps` (save intermediate files) switch directs the compiler to retain intermediate files generated, which are normally removed as part of the various compilation stages. These intermediate files are placed in the -path-output `directory` specified output directory or the build directory (when the `-path-output` switch is not used). See the *Input and Output File Extensions* table in Running the Compiler for a list of intermediate files.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > General > Save temporary files*.

-sdram

The `-sdram` (SDRAM is active) switch instructs the compiler to assume that at least Bank 0 of external SDRAM (the lower 32MB of space) is active and enabled. This switch is most useful for reducing the number of silicon anomaly workarounds needed. For more information, refer to Controlling Silicon Revision and Anomaly Workarounds Within the Compiler.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor > SDRAM Bank 0 is in use*.

-section *id=section_name[, id=section_name...]*

The `-section` switch controls the placement of types of data produced by the compiler. The data is placed into the *section_name* section as provided on the command line.

The compiler currently supports the following section identifiers (see Placement of Compiler-Generated Code and Data for more details).

Identifier	*Description*
`code`	Controls placement of machine instructions
`data`	Controls placement of initialized variable data
`constdata`	Controls placement of constant data
`bsz`	Controls placement of zero-initialized variable data
`sti`	Controls placement of the static C++ class constructor "start" functions. Default is `program`. For more information, see Constructors and Destructors of Global Class Instances.
`switch`	Controls placement of jump tables used to implement C/C++ switch statements. Default is `constdata`.
`vtbl`	Controls placement of the C++ virtual lookup tables
`vtable`	Synonym for `vtbl`
`strings`	Controls the placement of string literals
`autoinit`	Controls placement of data used to initialize aggregate autos
`alldata`	Controls placement of `data`, `constdata`, `bsz`, `strings`, and `autoinit` all at once

Note that `alldata` is not a real section kind, but rather a placeholder for `data`, `constdata`, `bsz`, `strings`, and `autoinit`.

Therefore,

```
-section alldata=X
```

is equivalent to:

```
-section data=X -section constdata=X -section bsz=X -section strings=X -section
autoinit=X
```

Ensure that the section selected via the command line exists within the `.ldf` file (refer to the *Linker and Utilities Manual*).

-show

The `-show` (display command line) switch shows the command-line arguments passed to `ccblkfn`, including expanded option files and environment variables. This allows you to ensure that command-line options have been passed successfully.

-signed-bitfield

The `-signed-bitfield` (make plain bit-fields signed) switch directs the compiler to make plain bit-field-those which have not been declared with an explicit `signed` or `unsigned` keyword-be signed. This is the default mode.

See also -unsigned-bitfield.

-signed-char

The `-signed-char` (make `char` signed) switch directs the compiler to make the default type for `char` signed. The compiler also defines the `__SIGNED_CHARS__` macro. This is the default mode when the -unsigned-char switch is not used.

-si-revision *version*

The `-si-revision` *version* (silicon revision) switch directs the compiler to build for a specific hardware revision (version). Any errata workarounds available for the targeted silicon revision will be enabled. For more information on valid revisions and the interactions of the `-si-revision`, `-workaround`, and `-no-workaround` switches, see Controlling Silicon Revision and Anomaly Workarounds Within the Compiler.

-structs-do-not-overlap

The `-structs-do-not-overlap` switch specifies that the source code being compiled contains no structure copies such that the source and the destination memory regions overlap each other in a non-trivial way.

For example, in the statement

```
*p = *q;
```

where `p` and `q` are pointers to some structure type `S`, the compiler, by default, always ensures that, after the assignment, the structure pointed to by `p` contains an image of the structure pointed to by `q` prior to the assignment. When `p` and `q` are not identical (in which case the assignment is trivial) but the structures pointed to by the two pointers may overlap each other, doing this means that the compiler must use the functionality of the C library function `memmove` rather than `memcpy`.

Using `memmove` to copy data is slower than using `memcpy`. Therefore, if your source code does not contain such overlapping structure copies, you can obtain higher performance by using the command-line switch `-structs-do-not-overlap`.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Language Settings > Structs/classes do not overlap*.

-syntax-only

The `-syntax-only` (only check syntax) switch directs the compiler to check the source code for syntax errors and warnings. No output files are generated with this switch.

-sysdefs

The `-sysdefs` (system macro definitions) switch directs the compiler to define several preprocessor macros describing the current user and user's system. The macros are defined as character string constants.

These macros are defined if the system returns information for them.

Table 2-14: System Macros Defined

Macro	Description
`__HOSTNAME__`	Name of the host machine
`__SYSTEM__`	Operating system name of the host machine
`__USERNAME__`	Current user's login name

-T *filename*

The `-T` *filename* (linker description file) switch directs the compiler (and linker) to use the specified linker description file (`.ldf`) as control input for linking. If `-T` is not specified, a default `.ldf` file is selected, based on the processor variant.

-threads

The `-threads` switch directs the compiler to link against the thread-safe variants of the C/C++ run-time libraries. The `-threads` switch defines the `_ADI_THREADS` macro as "1" at the compile, assemble, and link phases of a build.

> **NOTE:** The `-threads` switch does not imply that the compiler will produce thread-safe code when compiling C/C++ source. Make sure to use multi-threaded programming practices in code.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Linker > Processor > Link against thread-safe run-time libraries*.

See also -no-threads.

-time

The `-time` (tell time) switch directs the compiler to display elapsed time as part of the output information on each part of the compilation process.

-U *macro*

The `-U` *macro* (undefine macro) switch directs the compiler to undefine macros. If you specify a macro name, it is undefined. Note the compiler processes all `-D` (*define macro*) switches on the command line before any `-U` (*undefine macro*) switches.

NOTE: Add instances of this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Preprocessor > Preprocessor undefines*.

-unsigned-bitfield

The `-unsigned-bitfield` (make plain bit-fields unsigned) switch directs the compiler to make plain bit-fields, those which are declared with an explicit `signed` or `unsigned` keyword, unsigned.

Given the declaration,

```
struct {
    int a:2;
    int b:1;
    signed int c:2;
    unsigned int d:2;
} x;
```

the *Bit-Field Values* table lists the `bitfield` values.

Table 2-15: Bit-Field Values

Field	*-unsigned-bitfield*	*-signed-bitfield*	*Why*
x.a	-2..1	0..3	Plain field
x.b	0..1	-1..0	Plain field
x.c	-2..1	-2..1	Explicit signed
x.d	0..3	0..3	Explicit unsigned

See also -signed-bitfield.

-unsigned-char

The `-unsigned-char` (make `char` unsigned) switch directs the compiler to make the default type for `char` unsigned. The compiler also undefines the `__SIGNED_CHARS__` preprocessor macro.

-utility-rom

The `-utility-rom` (use Utility ROM) switch directs the tools to make use of the library routines in the processor's ROM, rather than retrieving versions from the libraries and linking them into the executable image. This can reduce the size of the final executable.

This switch is only supported for the ADSP-BF592-A and ADSP-BF7xx family processors. It is enabled by default when building for silicon revisions of those supported parts that can use their ROM.

See also -no-utility-rom.

-v

The `-v` (version and verbose) switch directs the compiler to display the version and command-line information for all the compilation tools as they process each file.

-verbose

The `-verbose` (display command line) switch directs the compiler to display command-line information for all the compilation tools as they process each file.

-version

The `-version` (display version) switch directs the compiler to display its version information.

-W{annotation|error|remark|suppress|warn} *number[, number...]*

The `-Wannotation`, `-Werror`, `-Wremark`, `-Wsuppress`, and `-Wwarn` (override error message) switches with a *number* argument direct the compiler to override the severity of the specified diagnostic messages (errors, remarks, or warnings). The *number* argument identifies the specific message to override.

At compilation time, the compiler produces a number for each specific compiler diagnostic message. A `{D}` (discretionary) following the diagnostic message number indicates that the diagnostic may have its severity overridden. Each diagnostic message is identified by a number that is used across all compiler software releases.

> **NOTE:** If the processing of the compiler command line generates a diagnostic, the position of the `-W` switch on the command-line is important. If the `-W` switch changes the severity of the diagnostic, it must occur before the command-line switch that generates the diagnostic; otherwise, no change of severity will occur.

Also, as shown in the Output view and in help, error codes sometimes begin with a leading zero (for example, cc0025). If you try to suppress error codes with `-W{annotation| error|remark|suppress|warn}` or `#pragma diag()` and supply the code with a leading zero, it will not work. This is because the compiler reads the number as an octal value, and will suppress a different warning or error.

-Wannotations

The `-Wannotations` (enable code generation annotations) switch directs the compiler to issue code generation annotations, which are messages milder than warnings that may help you to optimize your code.

> **NOTE:** Invoke this switch in the IDE by settings *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Warning > Warning/annotation/remark control* to *Errors, warnings and annotations*.

-Werror-limit *number*

The `-Werror-limit` *number* (maximum compiler errors) switch sets a maximum number of errors for the compiler before it aborts.

-Werror-warnings

The `-Werror-warnings` (treat warnings as errors) switch directs the compiler to treat all warnings as errors, with the result that a warning will cause the compilation to fail.

-Wremarks

The `-Wremarks` (enable diagnostic remarks) switch directs the compiler to issue remarks, which are diagnostic messages that are milder than warnings. Code generation annotations will also be issued, unless disabled with the `-no-annotate` switch (see -no-annotate).

NOTE: Invoke this switch in the IDE by settings *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Warning > Warning/annotation/remark control* to *Errors, warnings, annotations and remarks*.

-Wterse

The `-Wterse` (enable terse warnings) switch directs the compiler to issue the briefest form of warnings. This also applies to errors and remarks.

-w

The `-w` (disable all warnings) switch directs the compiler not to issue warnings.

NOTE: If the processing of the compiler command line generates a warning, the position of the `-w` switch on the command line is important. If the `-w` switch is located before the command-line switch that causes the warning, the warning is suppressed; otherwise, it is not suppressed.

NOTE: Invoke the switch in the IDE by settings *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Warning > Warning/annotation/remark control to Errors only*.

-warn-component

The `-warn-component` (warn if component elements are missing) switch instructs the compiler to issue warnings if it cannot locate libraries that are requested by the component's XML file. For more information, see -component file.xml.

-warn-protos

The `-warn-protos` (warn if incomplete prototype) switch directs the compiler to issue a warning when it calls a function for which an incomplete function prototype has been supplied. This option has no effect in C++ mode.

NOTE: Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Warning > Function declarations without prototypes*.

-workaround `workaround_id[, workaround_id]`

The `-workaround` `workaround_id[, workaround_id ...]` (enable avoidance of specific errata) switch enables compiler code generator workarounds for specific hardware errata. See Controlling Silicon Revision and Anomaly Workarounds Within the Compiler for details of valid workarounds and the interaction of the `-si-revision`, `-workaround`, and `-no-workaround` switches.

See also -no-workaround `workaround_id[, workaround_id]`.

-xref `filename`

The `-xref` `filename` (cross-reference list) switch directs the compiler to write cross-reference listing information to the specified file. When more than one source file has been compiled, the listing contains information about the last file processed.

For each reference to a symbol in the source program, a line of the following form is written to the named file.

Example:

```
symbol-id name ref-code filename line-number column-number
```

The `symbol-id` represents a unique decimal number for the symbol, and `ref-code` is one of the characters listed in the *ref-code Characters* table.

Table 2-16: ref-code Characters

Character	*Description*
D	Definition
d	Declaration
M	Modification
A	Address taken
U	Used
C	Changed (used and modified)
R	Any other type of reference
E	Error (unknown type of reference)

> **NOTE:** The compiler's `-xref` switch differs from the linker's `-xref` switch. Refer to the *Linker and Utilities Manual* for more information.

-zero-loop-counters

The `-zero-loop-counters` switch directs the compiler to ensure any used loop counters are set to zero on function exit. This switch should be used in the compilation of `initcode` that is overwritten with other code by an overlay manager or boot ROM that does not ensure loop counters are reset. Failure to do so may mean live hardware loops from `initcode` are encountered in the newly-loaded code, resulting in a random amount of loops over unrelated code (see *Hardware Loops* in the *Blackfin Processor Programming Reference* manual). Live hardware loops may be left when the compiler generates code that jumps out of a hardware loop before it reaches zero, for instance when generating an optimized `while` loop.

See also -no-zero-loop-counters.

C Mode (MISRA) Compiler Switch Descriptions

The following MISRA switches apply only to the C compiler. See MISRA-C Compiler for more information.

-misra

The `-misra` switch enables checking for MISRA-C Guidelines. Some rules or parts of rules are relaxed with this switch enabled. Rules relaxed by this option are 5.1, 5.7, 6.3, 6.4, 8.1, 8.2, 8.5, 10.5, 12.8, 13.7 and 19.7. This is explained in more detail, see Rule Descriptions.

The `-misra` switch is not supported in conjunction with some switches. For more information, see MISRA-C Command-Line Switch Restrictions. The switch predefines the `_MISRA_RULES` preprocessor macro.

-misra-linkdir *directory*

The `-misra-linkdir` switch specifies a directory in which to place `.misra` files. The default is a local directory called `MISRARepository`. The `.misra` files enable checking of violations of rules 5.5, 8.8, 8.9, and 8.10.

-misra-no-cross-module

The `-misra-no-cross-module` switch implies -misra, but also disables checking for a number of rules that require the use of the prelinker to check across multiple modules for rule violation. The MISRA-C rules suppressed are 5.5, 8.8, 8.9, and 8.10.

The `-misra-no-cross-module` switch is not supported in conjunction with some switches. For more information, see MISRA-C Command-Line Switch Restrictions.

-misra-no-runtime

The `-misra-no-runtime` switch implies -misra, but also disables run-time checking for MISRA-C rules 17.1, 17.2, 7.3, and 21.1. It limits the checking of rules 9.1, 12.8, 16.2, and 17.4.

The `-misra-no-runtime` switch is not supported in conjunction with some switches. For more information, see MISRA-C Command-Line Switch Restrictions.

-misra-strict

The `-misra-strict` switch enables checking for MISRA-C Guidelines. The switch ensures a strict interpretation of the MISRA-C:2004 Guidelines. See Rule Descriptions for more detail.

The `-misra-strict` switch is not supported in conjunction with some switches. For more information, see MISRA-C Command-Line Switch Restrictions. The switch predefines the `_MISRA_RULES` preprocessor macro.

-misra-suppress-advisory

The `-misra-suppress-advisory` switch implies `-misra`, but suppresses the reporting of advisory rules. The `-misra-suppress-advisory` switch is not supported in conjunction with some switches. For more information, see MISRA-C Command-Line Switch Restrictions.

-misra-testing

The `-misra-testing` switch implies `-misra` but also suppresses checking of MISRA-C rules 20.4, 20.7, 20.8, 20.9, 20.10, 20.11, and 20.12.

The `-misra-testing` switch is not supported in conjunction with some switches. For more information, see MISRA-C Command-Line Switch Restrictions.

-Wmis_suppress *rule_number[, rule_number]*

The `-Wmis_suppress` switch with a *rule_number* argument directs the compiler to suppress the specified diagnostic for a MISRA-C rule. The *rule_number* argument identifies the specific message to override

-Wmis_warn *rule_number[, rule_number]*

The `-Wmis_warn` switch with a *rule_number* argument directs the compiler to override the severity of the specified diagnostic to produce a warning for a MISRA-C rule. The *rule_number* argument identifies the specific message to override.

MISRA-C Command-Line Switch Restrictions

The following command-line switches are disallowed in MISRA-C mode.

- -w

- -W{annotation|error|remark|suppress|warn} *number[, number...]*

- -W{annotation|error|remark|suppress|warn} *number[, number...]*

- -c++

- -enum-is-int

- -warn-protos

- -decls-{weak|strong}

- -alttok

C++ Mode Compiler Switch Descriptions

The following switches apply only to the C++ compiler.

-anach

The -anach (enable C++ anachronisms) switch directs the compiler to accept some language features that are prohibited by the C++ standard but are still in common use. Use the -no-anach switch for greater standard compliance.

The following anachronisms are accepted when the -anach switch is enabled:

- Overload is allowed in function declarations. It is accepted and ignored.

- The number of elements in an array may be specified in an array delete operation. The value is ignored.

- A single operator++() function can be used to overload both prefix and postfix ++ operations.

- A single operator--() function can be used to overload both prefix and postfix -- operations.

- The base class name may be omitted in a base class initializer if there is only one immediate base class.

- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.

- A nested class name may be used as an un-nested class name provided no other class of that name has been declared. The anachronism is not applied to template classes.

- A reference to a non-const type may be initialized from a value of a different type. A temporary is created; it is initialized from the (converted) initial value, and the reference is set to the temporary.

- A reference to a non-const class type may be initialized from an rvalue of the class type or a derived class thereof. No (additional) temporary is used.

- A function with old-style parameter declarations is allowed and may participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when the check for compatibility is done, so that the following statements declare the overload of two functions named f:

```
int f(int);
int f(x) char x; { return x; }
```

See also -no-anach.

-check-init-order

It is not guaranteed that global objects requiring constructors are initialized before their first use in a program consisting of separately compiled units. The compiler outputs warnings if these objects are external to the compilation unit and are used in dynamic initialization or in constructors of other objects. These warnings are not dependent on the `-check-init-order` switch.

In order to catch uses of these objects and to allow the opportunity for code to be rewritten, the `-check-init-order` switch adds run-time checking to the code. This generates output to `stderr` to indicate that uses of such objects are unsafe.

> **NOTE:** This switch generates extra code to aid development. Do not use this switch when building production systems.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Language Settings > Check initialization order*.

-friend-injection

The `-friend-injection` switch directs the compiler to perform name lookup in a non-standard way with respect to friend declarations. With this switch enabled, a friend declaration is injected into the scope enclosing the class containing the friend declaration.

See also -no-friend-injection.

-full-cpplib

The `-full-cpplib` switch ensures the compilation uses the full ISO/IEC 14882:2003 standard library header files and library.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Language Settings > Use the Full C++ Standard Library and not the abridged library*.

This switch defines the `_ADI_FULLCPPLIB` macro during compilation and linking.

See also -no-full-cpplib.

> **NOTE:** Because certain operations in the full C++ standard library rely on wide-character I/O support, use of the full standard C++ library may require the full ANSI-compliant I/O libraries. Failure to use this library can result in the following linker error:
> ```
> [Error li1021] The following symbols referenced in processor 'p0'
>                could not be resolved:
>                'wide_character_IO_not_supported_without_full_io'
>                referenced from 'libio.dlb[widechar.doj]'
> ```
> To enable use of the full ANSI-compliant I/O library, do one of the following:

- Use the -full-io compiler switch.

- Set *I/O Libraries* to *Full ANSI C compliant I/O (-full-io)* in the IDE, on the *Project > Properties > C/C++ Build > Settings > Tool Settings > CrossCore Blackfin Linker > Libraries* page.

-full-dependency-inclusion

The `-full-dependency-inclusion` switch ensures that when generating dependency information for implicitly-included `.cpp` files, the `.cpp` file is re-included. This file is re-included only if the `.cpp` files are included more than once in the source (via re-inclusion of their corresponding header file). This switch is required only if your C++ sources files are compiled more than once with different macro guards.

NOTE: Enabling this switch may increase the time required to generate dependencies.

-implicit-inclusion

The `-implicit-inclusion` switch directs the compiler to enable the implicit inclusion of source files as a method of finding definitions of template entities to be instantiated. The compiler will automatically include a source file suffixed by `.C`, `.c`, or `.cpp` when the corresponding header file `.h` or `.hxx` is included.

See also -no-implicit-inclusion.

-no-anach

The `-no-anach` (disable C++ anachronisms) switch directs the compiler to disallow some old C++ language features that are prohibited by the C++ standard. This is the default mode. See the -anach switch for a full description of these features.

-no-friend-injection

The `-no-friend-injection` switch directs the compiler to conform to the ISO/IEC 14882:2003 standard with respect to friend declarations. The friend declaration is visible when the class to which it is a friend is among the associated classes considered by argument-dependent lookup. This is the default mode.

See also -friend-injection.

-no-full-cpplib

The `-no-full-cpplib` switch links the application with the abridged C++ library which consists of the embedded C++ library (EC++) and the standard template library (STL) as defined by the ISO/IEC 14882:2003 C++ standard. This switch is enabled by default.

-no-implicit-inclusion

The `-no-implicit-inclusion` switch prevents implicit inclusion of source files as a method of finding definitions of template entities to be instantiated. This is the default mode. This switch is accepted but ignored when compiling C files.

See also -implicit-inclusion.

-no-rtti

The `-no-rtti` (disable run-time type identification) switch directs the compiler to disallow support for `dynamic_cast` and other features of ANSI/ISO C++ run-time type identification. This is the default mode. Use `-rtti` to enable this feature.

See also -rtti.

-no-std-templates

The `-no-std-templates` switch disables dependent name processing (that is, the special lookup of names used in templates as required by the C++ standard).

See also -std-templates.

-rtti

The `-rtti` (enable run-time type identification) switch directs the compiler to accept programs containing `dynamic_cast` expressions and other features of ANSI/ISO C++ run-time type identification. The switch also causes the compiler to define the macro `__RTTI` to 1. See also the `-no-rtti` switch.

> **NOTE:** Invoke this switch in the IDE via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Language Settings > C++ exceptions and RTTI*.

See also -no-rtti.

-std-templates

The `-std-templates` switch enables dependent name processing, that is, the special lookup of names used in templates as required by the ISO/IEC 14882:2003 C++ standard. This is the default mode.

See also -no-std-templates.

Environment Variables Used by the Compiler

The compiler refers to several environment variables during its operation, as listed below. The majority of the environment variables identify *path names* to directories.

> **CAUTION:** Placing network paths into these environment variables may adversely affect the time required to compile applications.

- PATH

  This is your System search path, which is used to locate *binary executables* when you run them. The operating system uses this environment variable to locate the compiler when you execute it from the command line.

- TMP

  This directory is used by the compiler for temporary files, when building applications. For example, if you compile a C file to an object file, the compiler first compiles the C file to an assembly file which can be assembled to create the object file. The compiler usually creates a temporary directory within the TMP directory into which to put such files. However, if the `-save-temps` switch is specified, the compiler creates temporary

files in the current directory instead. This directory should exist and be writable. If this directory does not exist, the compiler issues a warning.

- `TEMP`

  This environment variable is also used by the compiler when looking for temporary files, but only if `TMP` was examined and was not set or the directory that `TMP` specified did not exist.

- `ADI_DSP`

  The compiler locates other tools in the tool-chain through the CCES installation directory, or through the `-path-install` switch. If neither is successful, the compiler looks in `ADI_DSP` for other tools.

- `CCBLKFN_OPTIONS`

  If this environment variable is set, and `CCBLKFN_IGNORE_ENV` is not set, this environment variable is interpreted as a list of additional switches to be prepended to the command line. Multiple switches are separated by spaces or new lines. A vertical-bar (`|`) character may be used to indicate that any switches following it will be processed after all other command-line switches.

- `CCBLKFN_IGNORE_ENV`

  If this environment variable is set, `CCBLKFN_OPTIONS` is ignored.

Additional Path Support

The compiler driver and compiler provide support for extensions to standard Windows pathnames. Both Windows shortcuts and Cygwin paths are supported. The extensions are controlled independently by compiler switches. Both features are disabled by default.

> **NOTE:** When either support is enabled, compilation time may be increased in cases where many include paths are passed to the compiler.

Windows Shortcut Support

Enable Windows shortcut support with the -expand-windows-shortcuts command-line switch, and disable it with the -no-expand-windows-shortcuts switch. The support is disabled by default. When enabled, the compiler recognizes elements of paths that refer to Windows shortcuts.

For example, if the source file `test.c` exists in the directory

```
c:\src\blackfin\
```

and a Windows shortcut is created as

```
c:\src\platform
```

which points to the source directory, the source file can be compiled with the command line:

```
ccblkfn -proc ADSP-BF533c:\src\platform\test.c -expand-windows-shortcuts
```

The compiler also recognizes path directory elements which are Windows shortcuts within preprocessor `#include` directives. For example, using the example above, a file containing

```
#include <platform\test.h>
```

could be compiled with the command line:

```
ccblkfn -proc ADSP-BF533 c:\src\platform\test.c -I c:\src -expand-windows-
shortcuts
```

Cygwin Path Support

The compiler provides support for Cygwin paths. The Cygwin environment provides users with a UNIX-like command-line environment on a Microsoft Windows machine.

> **NOTE:** The Cygwin environment is not part of CCES. It is provided by Red Hat, Inc. and can be downloaded from their website.

Cygwin path support is enabled with the `-expand-symbolic-links` switch and disabled with the `-no-expand-symbolic-links` switch. The support is disabled by default. The compiler recognizes three types of path extensions that are supported by Cygwin: symbolic links, cygdrive folders, and Cygwin mounted directories.

Cygwin Symbolic Links

Symbolic links are created within Cygwin using the `ln -s` command. The symbolic-links behave in a similar manner to Windows shortcuts, providing a secondary link to a file or directory.

For example, for the source file `test.c` located in directory `c:\src\blackfin\`, a symbolic link can be created using the commands:

```
cd \cygdrive\c\src ln -s platform blackfin
```

The source file can be compiled with the commands:

```
cd \cygdrive\c\src ccblkfn -proc ADSP-BF533 platform\test.c -expand-symbolic-links
```

> **NOTE:** The compiler supports local symbolic links only. CCES does not support symbolic links to remote devices and machines.

Cygdrive Folders

The Cygwin `\cygdrive` directory is a pseudo-directory that provides access to all the drives that can be located through the `My Computer` folder in Windows Explorer. The drives are accessed via the sub-directory corresponding to their drive letter.

For example, the `C:` drive is accessed via the directory `\cygdrive\c`, and the `c:\src\blackfin\test.c` file can be compiled using the command line:

```
ccblkfn -proc ADSP-BF533 \cygdrive\c\src\blackfin\test.c  -expand-symbolic-links
```

Cygwin Mounted Directories

Cygwin provides a `mount` command that reproduces the behavior of the UNIX `mount` command. It allows directories and devices to be accessed via an alternative "mounted" directory.

For example, to mount the directory `d:\testsuites` as `\tests`, issue the command:

```
mount d:\\testsuites \tests
```

The contents of `d:\testsuites` then is visible as if they existed within `\tests`. The file `d:\testsuites\test.c` can be compiled with the command:

```
ccblkfn -proc ADSP-BF533 \tests\test.c  -expand-symbolic-links
```

NOTE: The compiler supports local Cygwin mounts only. It does not support Cygwin mounts to remote devices and machines, nor does it support `\etc\fstab` mounts.

Optimization Control

The general aim of compiler optimization is to generate correct code that executes quickly and is small in size. Not all optimizations are suitable for every application or can be used all the time. Therefore, the compiler optimizer has a number of configurations, or optimization levels, which can be applied when needed. Each of these levels are enabled by one or more compiler switches (and CCES properties page) or pragmas.

NOTE: Refer to *Optimal Performance from C/C++ Source Code* for information on how to obtain maximal code performance from the compiler.

The compiler's optimization capabilities are described in the following sections.

Optimization Levels

The following list identifies several optimization levels. The levels are notionally ordered with least optimization listed first and most optimization listed last. The descriptions for each level outline the optimizations performed by the compiler and identify any required switches or pragmas that have direct influence on them.

- *Debug*

  The compiler produces debug information to ensure that the object code matches the appropriate source code line. For more information, see -g.

- *Default*

  The compiler does not perform any optimization by default when none of the compiler optimization switches are used (or enabled in the CCES *Properties* dialog box). Default optimization level can be enabled using the General Optimization Pragmas pragma.

- *Procedural Optimizations*

  The compiler performs advanced, aggressive optimization on each procedure in the file being compiled. The optimizations can be directed to favor optimizations for speed (-O1 or O) or space (-Os) or a factor between speed and space (-Ov). If debugging is also requested, the optimization is given priority so the debugging functionality may be limited. See -O[0|1], -Os, and -Ov *num* .

  Procedural optimizations for speed and space (-O and -Os) can be enabled in C/C++ source using the pragma `optimize_{for_speed|for_space}`. For more information, see General Optimization Pragmas.

- *Profile-Guided Optimizations (PGO)*

The compiler performs advanced aggressive optimizations using profiler statistics (.pgo files) generated from running the application using representative training data. PGO can be used in conjunction with interprocedural analysis (IPA) and automatic inlining. See -pguide for more information.

The most common scenario in collecting PGO data is to set up one or more simple file-to-device streams where the file is a standard ASCII stream input file and the device is any stream device supported by the simulator target, such as memory and peripherals. The PGO process can be broken down into the execution of one or more data sets where a data set is the association of zero or more input streams with one and only one .pgo output file.

For more information, see Using Profile-Guided Optimization in the *Optimal Performance from C/C++ Source Code* chapter.

> **NOTE:** Be aware of the requirement for allowing command-line arguments in your project when using PGO. For further details refer to Support for argv/argc.

- *Automatic Inlining*

  The compiler automatically inlines C/C++ functions which are not necessarily declared as inline in the source code. It does this when it has determined that doing so reduces execution time. The -Ov switch controls how aggressively the compiler performs automatic inlining. Automatic inlining is enabled using the -Oa switch which additionally enables procedural optimizations (-O). See -Oa, -Ov *num* , -O[0|1], and Function Inlining for more information.

  > **NOTE:** When remarks are enabled, the compiler produces a remark to indicate each function that is inlined.

- *Interprocedural Optimizations*

  The compiler performs advanced, aggressive optimization over the whole program, in addition to the per-file optimizations in procedural optimization. *Interprocedural analysis* (IPA) is enabled using the -ipa switch which additionally enables procedural optimizations (-O). See -ipa, -O[0|1], and Interprocedural Analysis for more information.

The compiler optimizer attempts to vectorize loops when it is safe to do so. IPA can identify additional safe candidates for vectorization which might not be classified as safe at a procedural optimization level. Additionally, there may be other loops that are known to be safe candidates for vectorization that can be identified to the compiler using various pragmas. (See Loop Optimization Pragmas.)

Using the various compiler optimization levels is an excellent way of improving application performance. However, consideration should be given to how applications are written so that compiler optimizations are given the best opportunity to be productive. These issues are the topic of *Optimal Performance from C/C++ Source Code.*

Interprocedural Analysis

The compiler has an optimization capability called *interprocedural analysis* (IPA) that allows the compiler to optimize across translation units instead of within individual translation units. This capability allows the compiler to see all of the source files used in a final link at compilation time and to use that information while optimizing.

Enable interprocedural analysis by selecting the *Interprocedural analysis* check box on the *Compile : General* page of the CCES *Properties* dialog box, or by specifying the -ipa command-line switch.

The -ipa switch automatically enables the -O switch to turn on optimization.

The -ipa switch generates additional files along with the object file produced by the compiler. These files have .ipa extensions and should not be deleted manually unless the associated object file is also deleted.

All of the -ipa optimizations are invoked after the initial link, when a special program called the prelinker re-invokes the compiler to perform the new optimizations, recompiling source files where necessary, to make use of gathered information.

> **NOTE:** Because a file may be recompiled by the prelinker, do not use the -S option to see the final optimized assembler file when -ipa is enabled. Instead, use the -save-temps switch, so that the full compile/link cycle can be performed first.

Interaction With Libraries

When IPA is enabled, the compiler examines all of the source files to build usage information about all of the function and data items. It then uses that information to make additional optimizations across all of the source files by recompiling where necessary.

Because IPA operates only during the final link, the -ipa switch has no benefit when initially compiling source files to object format for inclusion in a library. IPA gathers information about each file and embeds this within the object format, but cannot make use of it at this point, because the library contents have not yet been used in a specific context.

When IPA is invoked during linking, it will recover the gathered information from all linked-in object files that were built with -ipa, and where necessary and possible, will recompile source files to apply additional optimizations. Modules linked in from a library are not recompiled in this manner, as source is not available for them. Therefore, the gathered information in a library module can be used to further optimize application sources, but does not provide a benefit to the library module itself.

If a library module references a function in a user module in the program, this will be detected during the initial linking phase, and IPA will not eliminate the function. If the library module was not compiled with -ipa, IPA will not make any assumptions about how the function may be called, so the function may not be optimized as effectively as if all references to it were in source code visible to IPA, or from library modules compiled with -ipa.

Controlling Silicon Revision and Anomaly Workarounds Within the Compiler

The compiler provides three switches which specify that code produced by the compiler will be generated for a specific revision of a specific processor, and appropriate revision specific system run-time libraries will be linked against. Targeting a specific processor allows the compiler to produce code that avoids specific hardware errata reported against that revision. For the simplest control, use the -si-revision *version* switch, which automatically controls the use of compiler workarounds.

> **NOTE:** The compiler cannot apply errata workarounds to code inside asm() constructs.

When developing using the CCES IDE, the silicon revision used to build sources is part of a projects processor settings.

Using the -si-revision Switch

The `-si-revision` *version* (silicon revision) switch directs the compiler to build for a specific hardware revision. Any errata workarounds available for the targeted silicon revision will be enabled. The parameter `version` represents a silicon revision for the processor specified by the -proc *processor* switch. For example,

For example,

```
ccblkfn -proc ADSP-BF533 -si-revision 0.5 prog.c
```

If silicon version `any` is used, then errata workarounds are enabled for all supported revisions of the target processor.

If the `-si-revision` switch is not used, the compiler will default to target the latest known silicon revision for the target processor at the time of release, and any errata workarounds which are appropriate for the latest silicon revision will be enabled.

In the `Blackfin\lib` CCES installation directory there are a number of subdirectories. Within each of these is a complete set of libraries built for specific parts and silicon revisions. When linking an executable, the compiler driver selects and links against the best of these sets of libraries that is correct for the target part and has been built with the necessary silicon anomaly workarounds enabled to match the silicon revision switch. Note that an individual set of libraries may cover more than one specific part or silicon revision, so if several silicon revisions are affected by the same errata, then one common set of libraries might be used.

For processors other than ADSP-BF71x, the `__SILICON_REVISION__` macro is set by the compiler to two hexadecimal digits, representing the major and minor numbers in the silicon revision. For example, `1.0` becomes `0x100`, and `10.21` becomes `0xa15`. For ADSP-BF71x processors, the silicon revision is represented by a single letter, starting with A. The `__SILICON_REVISION__` macro is set by the compiler to a number based on the silicon revision letter: `0x0` for A, `0x1` for B, etc.

If the silicon revision is set to `any`, the `__SILICON_REVISION__` macro is set to `0xffff`.

The compiler driver will pass the `-si-revision` switch, as specified in the command line, when invoking other tools in the CCES tool chain.

> **NOTE:** Visit http://www.analog.com/processors/tools/anomalies for information on specific anomalies (including anomaly IDs).

Using the -workaround Switch

The -workaround *workaround_id[, workaround_id]* switch enables compiler code generator workarounds for specific hardware errata.

When workarounds are enabled, the compiler defines the macro `__WORKAROUNDS_ENABLED` at the compile, assembly, and link build stages. The compiler also defines individual macros for each of the enabled workarounds for each of these stages, as indicated by each macro description.

For a complete list of anomaly workarounds and associated `workaround_id` keywords, refer to the anomaly `.xml` files provided in the `<install_path>\System\ArchDef` directory. These are named in the format `<platform_name>-anomaly.xml`.

To find which workarounds are enabled for each chip and silicon revision, refer to the appropriate `<chip_name>-compiler.xml` file in the same directory (for example, `ADSP-BF533-compiler.xml`). Each `*-compiler.xml` file references an `*-anomaly.xml` file via the name in the `<cces-anomaly-dictionary>` element.

The anomaly `.xml` files relevant to Blackfin processors are `BLACKFIN-5xx-anomaly.xml` and `BLACKFIN-60x-anomaly.xml`.

> **NOTE:** Certain silicon anomalies affect the access of memory-mapped registers (MMRs), in particular 05-00-0122 (which is worked around by default), 05-00-0157 (under control of `-workaround killed-mmr-write`), and 05-00-0198 (under control of `-workaround sdram-mmr-read`). The compiler applies the appropriate workarounds to a memory access which it can identify as being to an MMR (for example, if the pointer to the MMR is assigned a literal address, or the value of the pointer can be calculated at compile time).
>
> For pointers whose destination may not be known until runtime, the compiler will take the conservative approach and assume that the pointer may access MMRs if it is volatile-qualified. To disable this assumption, use the -no-assume-vols-are-mmrs switch; the *memory-mapped register access functions* should be used to ensure the MMR access is made anomaly-safe.

Using the -no-workaround Switch

The `-no-workaround workaround_id[, workaround_id ...]` switch disables compiler code generator workarounds for specific hardware errata. For a list of valid workarounds, refer to the instructions in Using the -workaround Switch.

The `-no-workaround` switch can be used to disable workarounds enabled via the `-si-revision version` or `-workaround workaround_id` switches. All workarounds can be disabled by providing `-no-workaround` with all valid workarounds for the selected silicon revision or by using the option `-no-workaround all`. Disabling all workarounds via the `-no-workaround` switch will provide linking against libraries with no silicon revision in cases where the silicon revision is not `none`.

Interactions: Silicon Revision vs. Workaround Switches

Interactions between the `-si-revision`, `-workaround`, and `-no-workaround` switches can only be determined once all the command-line arguments have been parsed. To this effect, options are evaluated as follows:

1. The `-si-revision version` switch is parsed to determine which revision of the run-time libraries the application is to link against. It also produces an initial list of all the default compiler errata workarounds to enable.

2. Any additional workarounds specified with the `-workaround` switch is added to the errata list.

3. Any workarounds specified with `-no-workaround` is then removed from this list.

4. If any workarounds were declared via `-workaround`, the macro `__WORKAROUNDS_ENABLED` is defined at compile, assembly, and link stages, even if `-no-workaround` disables all workarounds.

Anomalies in Assembly Sources

If your project includes some hand-written assembly code, you have to ensure that you explicitly avoid any relevant anomalies that apply to your target processor. This can be simplified by the use of the `sys/anomaly_macros_rtl.h` header file. This header file defines macros for each of the anomalies that affect the run-time libraries, which allow for conditional inclusion of avoidance code.

For example, the following code makes use of the `WA_05000428` macro to conditionally select code that avoids problems with speculative reads from another core.

```
r3 = r1 ^ r2;
#if WA_05000428
    nop;
#endif
r6 = r2 +|+ r0 || r2 = [p2++] || nop;
```

Using Native Fixed-Point Types

This section provides an overview of the compiler's support for the native fixed-point types `fract` and `accum`, defined in Chapter 4 of the *Extensions to support embedded processors* ISO/IEC Technical Report 18037.

Fixed-Point Type Support

A fixed-point data type is one where the radix point is at a fixed position. This includes the integer types (the radix point is immediately to the right of the least-significant bit). However, this section uses the term to apply exclusively to those that have a non-zero number of fractional bits-that is, bits to the right of the radix point. There may also be integer bits to the left of the radix point.

The Blackfin processor has hardware support for arithmetic on a number of these fixed-point data types. For example, it is able to perform addition, subtraction and multiplication on 16-bit and 32-bit fractional values. However, the C language does not make it easy to express the semantics of the arithmetic that maps to the underlying hardware support.

To make it easier to use this hardware capability, and to facilitate expression of DSP algorithms that manipulate fixed-point data, the compiler supports a number of native fixed-point types whose arithmetic obeys the fixed-point semantics. This makes it easy to write high-performance algorithms that manipulate fixed-point data, without having to resort to compiler built-ins, or inline assembly.

An emerging standard for such fixed-point types is set out in Chapter 4 of the "*Extensions to support embedded processors*" ISO/IEC Technical Report 18037. CCES provides all the functionality specified in that chapter, and the chapter is a useful reference that explains the subtleties of the semantics of the library functions and arithmetic operators. However, the following sections give an overview of these data types, the semantics of arithmetic using these types, and guidelines for how to write high-performance code using these types.

Native Fixed-Point Types

Two keywords, _Fract and _Accum, are used to declare variables of fixed-point type. Each of these keywords may also be used in conjunction with the type specifiers short and long, and signed and unsigned. There are therefore 12 fixed-point types available, although some of these are aliases for types of the same size and format.

By including the header file stdfix.h, the more convenient alternative spellings - fract and accum - may be used instead of _Fract and _Accum. This header file also provides prototypes for many useful functions and it is highly recommended that you include it in source files that use fixed-point types. Therefore, the discussion that follows uses the spelling fract and accum as does the rest of the CCES documentation.

The formats of the fixed-point types are given in the ***Data Storage Formats, Ranges, and Sizes of the Native Fixed-Point Types*** table. In the ***Representation*** column of the table, the number after the point indicates the number of fractional bits, while the number before the point refers to the number of integer bits, including a sign bit when it is preceded by "s". Signed types are in two's complement form. The range of values that can be represented is also given in the table. Note that the bottom of the range can be represented exactly, whereas the top of the range cannot - only the value one bit less than this limit can be represented.

Table 2-17: Data Storage Formats, Ranges, and Sizes of the Native Fixed-Point Types

Type	*Representation*	*Range*	**sizeof** *Returns*
short fract	s1.15	[-1.0,1.0)	2
fract	s1.15	[-1.0,1.0)	2
long fract	s1.31	[-1.0,1.0)	4
unsigned short fract	0.16	[0.0,1.0)	2
unsigned fract	0.16	[0.0,1.0)	2
unsigned long fract	0.32	[0.0,1.0)	4
short accum	s9.31	[-256.0,256.0)	8
accum	s9.31	[-256.0,256.0)	8
long accum	s9.31	[-256.0,256.0)	8
unsigned short accum	8.32	[0.0,256.0)	8
unsigned accum	8.32	[0.0,256.0)	8
unsigned long accum	8.32	[0.0,256.0)	8

The Technical Report also defines a _Sat (alternative spelling sat) type qualifier for the fixed-point types. This stipulates that all arithmetic on fixed-point types shall be saturating arithmetic (that is, that the result of arithmetic that overflows the maximum value that can be represented by the type shall saturate at the largest or smallest representable value). When the sat qualifier is not used, the standard says that arithmetic that overflows may behave in an undefined manner. CCES accepts the sat qualifier for compatibility but will always produce code that saturates on overflow whether the sat qualifier is used or not. This gives maximum reproducibility of results and permits code to be written without worrying about obtaining unexpected results on overflow.

Native Fixed-Point Constants

Fixed-point constants may be specified in the same format as for floating-point constants, inclusive of any decimal or binary exponent. For more information on these formats, refer to strtofxfx in the *C/C++ Run-Time Library* chapter. Suffixes are used to identify the type of constants. The `stdfix.h` header also declares macros for the maximum and minimum values of the fixed-point types. See the *Fixed-Point Type Constant Suffixes and Macros* table for details of the suffixes and maximum and minimum fixed-point values.

Table 2-18: Fixed-Point Type Constant Suffixes and Macros

Type	Suffix	Example	Minimum Value	Maximum Value
short fract	hr	0.5hr	SFRACT_MIN	SFRACT_MAX
fract	r	0.5r	FRACT_MIN	FRACT_MAX
long fract	lr	0.5lr	LFRACT_MIN	LFRACT_MAX
unsigned short fract	uhr	0.5uhr	0.0uhr	USFRACT_MAX
unsigned fract	ur	0.5ur	0.0ur	UFRACT_MAX
unsigned long fract	ulr	0.5ulr	0.0ulr	ULFRACT_MAX
short accum	hk	12.4hk	SACCUM_MIN	SACCUM_MAX
accum	k	12.4k	ACCUM_MIN	ACCUM_MAX
long accum	lk	12.4lk	LACCUM_MIN	LACCUM_MAX
unsigned short accum	uhk	12.4uhk	0.0uhk	USACCUM_MAX
unsigned accum	uk	12.4uk	0.0uk	UACCUM_MAX
unsigned long accum	ulk	12.4ulk	0.0ulk	ULACCUM_MAX

A Motivating Example

Consider a very simple example, a fixed-point dot product. How do you write it using the native fixed-point types? The algorithm performs multiplication of each pair of fractional values in the input arrays. The `accum` type is designed to hold the results of accumulations, which is exactly what is needed. Assume that the data consist of vectors of 16-bit values, representing values in the range [-1.0,1.0). Then it is natural to write:

```
#include <stdfix.h>

accum dot_product(fract *a, fract *b, int n)
{
    accum sum = 0.0k;
    int i;
    for (i = 0; i<n; i++)
        sum += a[i] * b[i];
    return sum;
}
```

The above algorithm performs a pair-wise fractional multiplication of elements of the input arrays and accumulates the result into a variable that saturates on overflow. In fact, this simple expression of the algorithm hides a subtlety related to the semantics of the arithmetic which is discussed in FX_CONTRACT, but it does show that it is easy to express algorithms that manipulate fixed-point data and perform saturation on overflow without needing to find special ways to express these semantics through integer arithmetic.

Fixed-Point Arithmetic Semantics

The semantics of fixed-point arithmetic according to the Technical Report are as follows:

1. If a binary operator has one floating-point operand, the other operand is converted to floating-point and the operator is applied to two floating-point operands to give a floating-point result.

2. If the operator has two fixed-point operands of different signedness, convert the unsigned one to signed without changing its size. (However, see also FX_CONTRACT.)

3. Deduce the result type. The result type is the operand type of highest rank. Rank increases in the following order: `short fract`, `fract`, `long fract`, `short accum`, `accum`, `long accum` (or their unsigned equivalents). An operator with only one fixed-point operand produces a result of this fixed-point type. (An exception is the result of a comparison, which gives a boolean result.)

4. The result is the mathematical result of applying the operator to the operand values, converted to the result type deduced in step 3. In other words, the result is as if it was computed to infinite precision before converting this result to the final result type.

The conversions between different types are discussed in Data Type Conversions and Fixed-Point Types.

Data Type Conversions and Fixed-Point Types

The rules for conversion to and from fixed-point types are as follows:

1. When converting to a fixed-point type, if the value of the operand can be represented by the fixed-point type, the result is this value. If the operand value is out of range of the fixed-point type, the result is the closest fixed-point value to the operand value. In other words, conversion to fixed-point saturates the operand's mathematical value to the fixed-point type's range. If the operand value is within the range of the fixed-point type, but cannot be represented exactly, the result is the closest value either higher or lower than the operand value. For more information, see Rounding Behavior.

2. When converting to an integer type from a fixed-point type, the result is the integer part of the fixed-point type. The fractional part is discarded, so rounding is towards zero; `(int)(1.9k)` gives 1, and `(int)(-1.9k)` gives -1.

3. When converting to a floating-point type, the result is the closest floating-point value to the operand value.

These rules have some important consequences of which you should be aware:

CAUTION: Conversion of an integer to a fractional type is only useful when the integer is -1, 0, or 1. Any other integer value will be saturated to the fractional type.

So a statement similar to

```
fract f = 0x4000; // try to assign 0.5 to f
```

does not assign `0.5` to `f`, but instead results in `FRACT_MAX` because `0x4000` is an integer greater than 1. Instead, use

```
fract f = 0.5r;
```

- or -

```
fract f = 0x4000p-15r;
```

Note that the second format above uses the binary exponent syntax available for fixed-point constants; specifically the value `0x4000` is scaled by 2^{-15}.

CAUTION: Assignment of a fractional value to an integer yields zero unless the fractional value is -1.0. Assignment of an unsigned fractional value to an integer always results in zero.

CAUTION: Be very careful to avoid mixing `fract16` and `fract32` types with `fract` and `long fract`. The former are typedefs to integer types; thus:

```
#include <stdfix.h>
#include <fract.h>
fract16 f16;
fract f;

void foo(void) {
    f16 = -0x4000;    // stores -0.5 into f16
    f = f16;          // gives f = -1.0
}
```

because `f16` is an integer value and therefore saturates on assignment to the true fractional type. The compiler emits an error when it can detect that a `fract16` or `fract32` value has been converted to a `fract` or `long fract` type (or vice versa), because this nearly always indicates a programming error. To convert between the integer typedefs and the native types, use Bit-Pattern Conversion Functions: bitsfx and fxbits.

Compiler warnings are produced to aid in the diagnosis of problems where these conversions are likely to produce unexpected results.

Bit-Pattern Conversion Functions: bitsfx and fxbits

The `stdfix.h` header file provides functions to convert a bit pattern to a fixed-point type and vice versa. These functions are particularly useful for converting between native types (`fract`, `long fract`) and integer typedefs (`fract16`, `fract32`).

For each fixed-point type, a corresponding integer type is declared, which is big enough to hold the bit pattern for the fixed-point type. These are `int_fx_t`, where *fx* is one of hr, r, lr, hk, k, or lk, and `uint_fx_t` where *fx* is one of uhr, ur, ulr, uhk, uk, or ulk.

To convert a fixed-point type to a bit pattern, use the `bitsfx` family of functions. *fx* may be any of hr, r, lr, hk, k, lk, uhr, ur, ulr, uhk, uk, or ulk. For example, using the prototype

```
uint_ur_t bitsur(unsigned fract);
```

you can write

```
#include <stdfix.h>
unsigned fract f;
uint_ur_t f_bit_pattern;

void foo(void) {
   f = 0.5ur;
   f_bit_pattern = bitsur(f);    // gives 0x8000
}
```

> **NOTE:** This is a good way to convert from a `fract` to a `fract16` or a `long fract` to a `fract32` where necessary. For example,

```
#include <stdfix.h>
#include <fract.h>
fract f;
fract16 f16;

void foo(void) {
    f = 0.5r;
    f16 = bitsr(f);     // 0x4000 as expected
}
```

For more information, see bitsfx in the *C/C++ Run-Time Library* chapter.

Similarly, to convert to a fixed-point type from a bit pattern, use the `fxbits` family of functions. So, to convert from a `fract32` to a `long fract`, use:

```
#include <stdfix.h>
#include <fract.h>
fract32 f32;
long fract lf;

void foo(void) {
    f32 = 0x40000000;     // that's 0.5
    lf = lrbits(f32);     // gets 0.5lr as expected
}
```

For more information, see fxbits in the *C/C++ Run-Time Library* chapter.

Arithmetic Operators for Fixed-Point Types

You can use the +, −, *, and / operators on fixed-point types, which have the same meaning as their integer or floating-point equivalents, aside from any overflow or rounding semantics. As discussed in Native Fixed-Point Types, fixed-point operations that overflow give results saturated at the highest or lowest fixed-point value. Rounding is discussed in Rounding Behavior.

You can use << to shift a fixed-point value up by a positive integer shift amount less than the fixed-point type size in bits. This gives the same result as multiplication by a power of 2, including overflow semantics:

```
#include <stdfix.h>
fract f1, f2;

void foo1(void) {
   f1 = 0.125r;
   f2 = f1 << 2;      // gives 0.5r
}

void foo2(void) {
   f1 = -0.125r;
   f2 = f1 << 10;     // gives -1.0r
}
```

You can also use >> to shift a fixed-point value down by an integer shift amount in the same range. This is defined to give the same result as division by a power of 2, including any rounding behavior:

```
#include <stdfix.h>
fract f1, f2;

void foo1(void) {
   f1 = 0.5r;
   f2 = f1 >> 2;      // gives 0.125r
}

void foo2(void) {
   f1 = 0x0003p-15r;
   f2 = f1 >> 2;         // gives 0x0000p-15r when rounding mode
                         // is truncation
                         // and 0x0001p-15r when rounding mode
                         // is biased or unbiased
 }
```

Any of these operators can be used in conjunction with assignment, For example,

```
#include <stdfix.h>
fract f1, f2;

void foo1(void) {
   f1 = 0.2r;
   f2 = 0.3r;
   f2 += f1;
}
```

In addition, there are a number of unary operators that may be used with fixed-point types. These are:

- ++ Equivalent to adding integer 1

- -- Equivalent to subtracting integer 1

- + Unary plus, equivalent to adding value to 0.0 (no effect)

- – Unary negate, equivalent to subtracting value from 0.0

- ! 1 if equal to 0.0, 0 otherwise

FX_CONTRACT

The example of a dot product (see A Motivating Example) contains the accumulation:

```
sum += a[i] * b[i];
```

where sum is an `accum` type, and `a[i]`, `b[i]` are `fract` types. Bearing in mind the rules discussed in the previous section, what is the result of the multiplication? Since both `a[i]` and `b[i]` are `fract` types, the result of the multiplication is also a `fract` - in other words, two s1.15 operands are multiplied together to yield an s1.15 result. So the rules say that it should be equivalent to writing:

```
fract tmp = a[i] * b[i];
sum += tmp;
```

However, this means that:

- The multiply result must be rounded to s1.15; 15 bits of precision are lost.

- The result of multiplying `-1.0r` by `-1.0r` should be `FRACT_MAX` - that is, not quite 1.0.

There are two problems with this:

- You probably do not want to round away those extra bits of precision before adding the result of the multiplication to `sum`. Doing so decreases the accuracy of the accumulation. Moreover, the Blackfin processor has an efficient single-cycle multiply-accumulate instruction, but this does not discard the extra bits of precision in the multiply result before accumulation.

- On Blackfin processors, the multiply-accumulate instruction does not saturate `-1.0r * -1.0r` before adding to the accumulator register. This again has the effect of increasing the accuracy of the accumulated result, but does not match the fixed-point type semantics for the dot product example.

To generate efficient code without losing precision, you should really write:

```
sum += (accum)a[i] * (accum)b[i];
```

This is because the conversion to the higher-precision `accum` type prior to multiplication means that the generated code can hold the intermediate multiply result in s9.31 format, which means there is no requirement to saturate the result or round off the lower order bits. This allows the compiler to use the hardware multiply-accumulate instruction.

For convenience, the compiler can do this step for you, using a mode known as `FX_CONTRACT`. The name `FX_CONTRACT` is used as the behavior is similar to that of `FP_CONTRACT` in C99. When `FX_CONTRACT` is on, the compiler may keep intermediate results in greater precision than that specified by the Technical Report. In other words, it may choose not to round away extra bits of precision or to saturate an intermediate result unnecessarily. More precisely, the compiler keeps the intermediate result in greater precision when:

- Maintaining the higher-precision intermediate result will be more efficient - it maps better to the underlying hardware.

- The intermediate result is not stored back to any named variable.

- No explicit casts convert the type of the intermediate result.

In other words,

```
sum += a[i] * b[i];
```

results in a multiply-accumulate instruction, but

```
sum += (fract)(a[i] * b[i]);
```

- or -

```
fract tmp = a[i] * b[i];
sum += tmp;
```

both force the result of the multiply to be converted back to `fract` type before the accumulation.

There are other examples where `FX_CONTRACT` may keep intermediate results in higher precision:

- Implicit conversion of unsigned fixed-point type to a larger signed fixed-point type does not first convert to the signed fixed-point type of the smaller size.

- Multiplication of `signed fract` and `unsigned fract` can create a mixed-mode fractional multiply rather than first converting the `unsigned fract` to a `signed fract`.

By default, the compiler permits `FX_CONTRACT` behavior. The `FX_CONTRACT` mode can be controlled with a pragma (see #pragma FX_CONTRACT {ON|OFF}) or with command-line switches, `-fx-contract` and `-no-fx-contract` (see -fx-contract and -no-fx-contract). The pragma may be used at file scope or within functions. It obeys the same scope rules as the `FX_ROUNDING_MODE` pragma discussed with an example in Setting the Rounding Mode.

Rounding Behavior

What happens if a `long fract` is converted to a `fract`? The 16 least-significant bits cannot be represented in the result, so they must be discarded during the conversion. In the case where the `long fract` value cannot be represented exactly by the `fract` type, there is a choice: the result can be the nearest `fract` value greater than the `long fract` value, or the nearest value less than the `long fract` value. This is known as the rounding behavior.

Some fixed-point operations are also affected by rounding. For example, multiplication of two fractional values to produce a fractional result of the same size requires discarding a number of bits of the exact result. For example, s1.15 * s1.15 produces an exact s2.30 result. This is saturated to s1.30 and the fifteen least-significant bits must be discarded to produce an s1.15 result.

By default, any bits that must be discarded are truncated-in other words, they are simply chopped off the end of the value. For example:

```
#include <stdfix.h>
fract f1, f2, prod;

void foo(void) {
   f1 = 0x3ffp-15r;
   f2 = 0x1000p-15r;
   prod = f1 * f2;   // gives 0x007fp-15r, discarded
                     // least-significant bits 0xe000
}
```

This is equivalent to always rounding down toward negative infinity. It tends to produce results whose accuracy tends to deteriorate as any rounding errors are generally in the same direction and are compounded as the calculations proceed.

If this does not give you the accuracy you require, you can use either biased or unbiased round-to-nearest rounding. The compiler supports pragmas and switches to control the rounding mode. In the biased or unbiased rounding modes, the above product will be rounded to the nearest value that can be represented by the result type, so the final result will be `0x0080p-15r`.

The difference between biased and unbiased rounding occurs when the value to be rounded lies exactly half-way between the two closest values that can be represented by the result type. In this case, biased rounding will always round toward the greater of the two values (applying saturation if this rounding overflows) whereas unbiased rounding will round toward the value whose least-significant bit is zero. For example,

```
#include <stdfix.h>
fract f;
long fract lf;

void foo1(void) {
   lf = 0x34568000p-31lr;
   f = lf;   // gives 0x3456p-15r in unbiased rounding mode,
             // but 0x3457p-15r in biased rounding mode
}

void foo2(void) {
   lf = 0x34578000p-31lr;
   f = lf;        // gives 0x3458p-15r in both biased
                  // and unbiased rounding modes
}
```

In general, unbiased rounding is more costly than biased rounding in terms of cycles, but yields a more accurate result since rounding errors in the half-way case are not all in the same direction and therefore are not compounded so strongly in the final result.

The rounding discussed here only affects operations that yield a fixed-point result. Operations that yield an integer result round toward zero. There are also a few exceptions to the rounding rules:

- Conversion of a floating-point value to a fixed-point value rounds towards zero.

- The `roundfx`, `strtofxfx`, and `fxdivi` functions always perform either biased or unbiased rounding, dependent on the current state of the `RND_MOD` bit. They do not support the truncation rounding mode.

Details of how to set rounding mode are given in Setting the Rounding Mode.

Arithmetic Library Functions

The `stdfix.h` header file also declares a number of functions that permit useful arithmetic operations on combinations of fixed-point and integer types. These are the `divifx`, `idivfx`, `fxdivi`, `mulifx`, `absfx`, `roundfx`, `countlsfx`, and `strtofxfx` families of functions.

divifx

The `divifx` functions, where *fx* is one of r, lr, k, lk, ur, ulr, uk, or ulk, allow division of an integer value by a fixed-point value to produce an integer result. If you write

```
#include <stdfix.h>
fract f;
int i, quo;

void foo(void) {
   // BAD: division of int by fract gives fract result, not int
   f = 0.5r;
   i = 2;
   quo = i / f;
}
```

then the result of the division is a `fract` whose integer part is stored in the variable `quo`. This means that the value of `quo` is zero, as the division overflows and thus produces a fractional result that is nearly one.

To get the desired result, write

```
#include <stdfix.h>
fract f;
int i, quo;

void foo(void) {
   // GOOD: uses divifx to give integer result
   f = 0.5r;
   i = 2;
   quo = divir(i, f);
}
```

which stores the value 4 into the variable `quo`.

For more information, see the functional description divifx in the *C/C++ Run-Time Library* chapter.

idivfx

The `idivfx` functions, where *fx* is one of r, lr, k, lk, ur, ulr, uk, or ulk, allow division of a fixed-point value by a fixed-point value to produce an integer result. If you write

```
#include <stdfix.h>
fract f1, f2;
int quo;

void foo(void) {
   // BAD: division of two fracts gives fract result, not int
   f1 = 0.5r;
   f2 = 0.25r;
   quo = f1 / f2;
}
```

then the result of the division is a `fract` whose integer part is stored in the variable `quo`. This means that the value of `quo` is zero, as the division overflows and thus produces a fractional result that is nearly one.

To get the desired result, write

```
#include <stdfix.h>
fract f1, f2;
int quo;

void foo(void) {
   // GOOD: uses idivfx to give integer result
   f1 = 0.5r;
   f2 = 0.25r;
   quo = idivr(f1, f2);
}
```

which stores the value 2 into the variable `quo`.

For more information, see the idivfx functional description in the *C/C++ Run-Time Library* chapter.

fxdivi

The `fxdivi` functions, where *fx* is one of r, lr, k, lk, ur, ulr, uk, or ulk, allow division of an integer value by an integer value to produce a fixed-point result. If you write

```
#include <stdfix.h>
int i1, i2;
fract quo;

void foo(void) {
   // BAD: division of int by int gives int result, not fract
   i1 = 5;
   i2 = 10;
   quo = i1 / i2;
}
```

then the result of the division is an integer which is then converted to a `fract` to be stored in the variable `quo`. This means that the value of `quo` is zero, as the division is rounded to integer zero and then converted to `fract`.

To get the desired result, write

```
#include <stdfix.h>
int i1, i2;
fract quo;

void foo(void) {
   // GOOD: uses fxdivi to give fract result
   i1 = 5;
   i2 = 10;
   quo = rdivi(i1, i2);
}
```

which stores the value 0.5 into the variable `quo`.

For more information, see the functional description fxdivi in the *C/C++ Run-Time Library* chapter.

mulifx

The `mulifx` functions, where `fx` is one of r, lr, k, lk, ur, ulr, uk, or ulk, allow multiplication of an integer value by a fixed-point value to produce an integer result. If you write

```
#include <stdfix.h>
int i, prod;
fract f;

void foo(void) {
        // BAD: multiplication of int by fract
        // produces fract result, not int
   i = 50;
   f = 0.5r;
   prod = i * f;
}
```

then the result of the multiplication is a `fract` whose integer part is stored in the variable `prod`. This means that the value of `prod` is zero, as the multiplication overflows and thus produces a fractional result that is nearly one.

To get the desired result, write

```
#include <stdfix.h>
int i, prod;
fract f;

void foo(void) {
        // GOOD: uses mulifx to give integer result
   i = 50;
   f = 0.5r;
   prod = mulir(i, f);
}
```

which will store the value 25 into the variable `prod`.

For more information, see the mulifx functional description in the *C/C++ Run-Time Library* chapter.

absfx

The `absfx` functions, where *fx* is one of hr, r, lr, hk, k, or lk, compute the absolute value of a fixed-point value.

In addition, you can also use the type-generic macro `absfx()`, where the operand type can be any of the signed fixed-point types.

For more information, see the functional description absfx in the *C/C++ Run-Time Library* chapter.

roundfx

The `roundfx` functions, where *fx* is one of hr, r, lr, hk, k, lk, uhr, ur, ulr, uhk, uk, or ulk, take two arguments. The first is a fixed-point operand whose type corresponds to the name of the function called. The second gives a number of fractional bits. The first operand is rounded to the number of fractional bits given by the second operand. The second operand must specify a value between 0 and the number of fractional bits in the type. Rounding is to-nearest. However, whether the rounding is biased or unbiased depends on the state of the RND_MOD bit on the hardware. See Rounding Behavior for details.

```
#include <stdfix.h>
long fract lf, rnd;

void foo1(void) {
   lf = 0x45608100p-31lr;
   rnd = roundlr(lf, 15); // produces 0x45610000p-31lr;
}

void foo2(void) {
   lf = 0x7fff9034p-31lr;
   rnd = roundlr(lf, 15); // produces 0x7fffffffp-31lr;
}
```

In addition, you can also use the type-generic macro `roundfx()`, where the first operand type can be any of the signed fixed-point types.

For more information, see the functional description roundfx in the *C/C++ Run-Time Library* chapter

countlsfx

The `countlsfx` functions, where *fx* is one of hr, r, lr, hk, k, lk, uhr, ur, ulr, uhk, uk, or ulk, return the largest integer value k such that its operand, when shifted up by k, does not overflow. For zero input, the result is the size in bits of the operand type.

```
#include <stdfix.h>
int scal1, scal2;

void foo(void) {
   scal1 = countlsk(-3.0k);    // gives 6, because
                               // -3.0k<<6 = -192.0k
   scal2 = countlsuk(3.0uk);   // gives 6, because
                               // 3.0uk<<6 = 192.0uk
}
```

In addition, you can also use the type-generic macro `countlsfx()`, where the operand type can be any of the signed fixed-point types.

For more information, see the function description countlsfx in the *C/C++ Run-Time Library* chapter.

strtofxfx

The `strtofxfx` functions, where *fx* is one of hr, r, lr, hk, k, lk, uhr, ur, ulr, uhk, uk, or ulk, parse a string representation of a fixed-point number and return a fixed-point result. They behave similarly to `strtod` and accept input in the same format.

For more information, see the strtofxfx functional description in the *C/C++ Run-Time Library* chapter.

I/O Conversion Specifiers

The `printf` and `scanf` families of functions support conversion specifiers for the fixed-point types. These are given in the *I/O Conversion Specifiers for Fixed-Point Types* table. Note that the conversion specifiers for the signed types, `%r` and `%k`, are lowercase while those for the unsigned types, `%R` and `%K`, are uppercase.

Table 2-19: I/O Conversion Specifiers for Fixed-Point Types

Type	Conversion Specifier
short fract	%hr
fract	%r
long fract	%lr
unsigned short fract	%hR
unsigned fract	%R
unsigned long fract	%lR
short accum	%hk
accum	%k
long accum	%lk
unsigned short accum	%hK
unsigned accum	%K
unsigned long accum	%lK

When used with the `scanf` family of functions, these conversion specifiers accept input in the same format as consumed by the `strtofxfx` functions, which is the same as that accepted for `%f`. For more information, see the strtofxfx functional description in the *C/C++ Run-Time Library* chapter.

When used with the `printf` family of functions, fixed-point values are printed:

- As hexadecimal values by default, or when the `-no-full-io` compiler switch is used. For example,

```
printf("fract: %r\n", 0.5r);  // prints   fract: 4000
```

- Like floating-point values when the `-fixed-point-io` or `-full-io` compiler switches are used. For example,

```
printf("fract: %r\n", 0.5r);   // prints   fract: 0.500000
```

Optional precision specifiers are accepted that control the number of decimal places printed, and whether a trailing decimal point is printed. However, these have no effect unless either `-fixed-point-io` or `-full-io` are used. For more information, see fprintf in the *C/C++ Run-Time Library* chapter.

Setting the Rounding Mode

As discussed in Rounding Behavior, there are three rounding modes supported for fixed-point arithmetic:

- Truncation (this is the default rounding mode)

- Biased round-to-nearest rounding

- Unbiased round-to-nearest rounding

To set the rounding mode, you can use a pragma or a compile-time switch.

The following compile-time switches control rounding behavior:

- -fx-rounding-mode-truncation

- -fx-rounding-mode-biased

- -fx-rounding-mode-unbiased

The given rounding mode will then be the default for the whole of the source file being compiled.

You can also use a pragma to allow finer-grained control of rounding. The pragmas are:

- `#pragma FX_ROUNDING_MODE TRUNCATION`

- `#pragma FX_ROUNDING_MODE BIASED`

- `#pragma FX_ROUNDING_MODE UNBIASED`

If one of these pragmas is applied at file scope, it applies until the end of the translation unit or until another pragma at file scope changes the rounding mode.

If one of these pragmas is applied within a compound statement (that is, within a block enclosed by braces), the pragma applies to the end of the compound statement where it is specified. The rounding mode returns to the outer scope rounding mode on exit from the compound statement. An example of how to use these pragmas is given in the following example.

Use of #pragma FX_ROUNDING_MODE to Control Rounding of Arithmetic on Fixed-Point Types

```
#include <stdfix.h>
#pragma FX_ROUNDING_MODE BIASED

fract my_func(void) {
   // rounding mode here is biased
```

```
    {
        #pragma FX_ROUNDING_MODE UNBIASED
        // rounding mode here is unbiased
    }
      // rounding mode here is biased
}
#pragma FX_ROUNDING_MODE TRUNCATION
fract my_func2(void) {
        // rounding mode here is truncation
}
```

Blackfin has specialized instructions to support round-to-nearest rounding. However, whether these perform biased or unbiased rounding is dependent on the current state of the RND_MOD bit. In order to facilitate generation of efficient code, the compiler will assume that when the rounding mode is either biased or unbiased, the RND_MOD bit has been set to the same type of rounding. This means that the compiler can use the hardware support for these rounding modes efficiently without needing to set or clear this bit every time it uses a RND_MOD bit-dependent instruction.

Thus, it is your responsibility to ensure that the RND_MOD bit is set correctly. Built-in functions are provided to make this task easier:

- int set_rnd_mod_biased(void)

- int set_rnd_mod_unbiased(void)

The return value of these built-in functions is the previous state of the RND_MOD bit. So, another built-in function (void restore_rnd_mod(int)) resets the RND_MOD bit to a saved value.

For example, you could write:

```
#include <stdfix.h>
#include <builtins.h>

fract my_func(void) {
    #pragma FX_ROUNDING_MODE BIASED
    int saved_rnd_mod = set_rnd_mod_biased();
    // rounding mode now biased
    restore_rnd_mod(saved_rnd_mod);
    // rounding mode now same as on function entry
}
```

If you use the pragmas to specify biased or unbiased rounding without setting the RND_MOD bit, you may get a mixture of biased and unbiased rounding behavior.

For more information, see #pragma FX_ROUNDING_MODE {TRUNCATION|BIASED|UNBIASED} and Changing the RND_MOD Bit.

Porting Code Written Using fract16 and fract32

If you have code written using `fract16` and `fract32` types, along with built-in functions and calls to library functions, you may wish to rewrite your code to use the new native fixed-point types. This section contains a number of tips for the easiest ways to do that.

Since `fract` is a 16-bit type and `long fract` is a 32-bit type, the basic strategy will be to replace uses of `fract16` variables with `fract`-typed ones, and `fract32` variables with `long fract`-typed ones.

Firstly, code written using `fract16` and `fract32` will often contain constants. If these are written using the `r16` and `r32` suffixes, you can simply change the suffix to create a native fixed-point type.

For example,

```
fract16 f1 = 0.5r16;
fract32 f2 = 0.75r32;
```

becomes

```
fract f1 = 0.5r;
long fract f2 = 0.75lr;
```

If your code contains hexadecimal constants, it is convenient to use the binary exponent syntax to convert your constants:

```
fract16 f1 = 0x1234;
fract32 f2 = 0x12345678;
```

becomes

```
fract f1 = 0x1234p-15r;
long fract f2 = 0x12345678p-31lr;
```

Many built-ins are no longer necessary once you have converted to the native fixed-point types-you can use native arithmetic instead. The correspondence between the `fract16` and `fract32` built-in functions and native fixed-point arithmetic is given in the ***Correspondence Between fract16 and fract32 Built-In Functions and Native Fixed-Point Arithmetic*** table.

Table 2-20: Correspondence Between fract16 and fract32 Built-In Functions and Native Fixed-Point Arithmetic

fract16 or fract32 Built-In Function	*Native Fixed-Point Type Arithmetic*
`fract16 f1, f2;` `fract16 f3 = add_fr1x16(f1, f2);`	`fract f1, f2;` `fract f3 = f1+ f2;`
`fract16 f1, f2;` `fract16 f3 = sub_fr1x16(f1, f2);`	`fract f1, f2;` `fract f3 = f1- f2;`
`fract16 f1, f2;` `fract16 f3 = mult_fr1x16(f1, f2);`	`fract f1, f2;` `fract f3 = f1* f2; // in truncation rounding mode`

Table 2-20: Correspondence Between fract16 and fract32 Built-In Functions and Native Fixed-Point Arithmetic (Continued)

fract16 or fract32 Built-In Function	*Native Fixed-Point Type Arithmetic*
`fract16 f1, f2;` `fract16 f3 = multr_fr1x16(f1, f2);`	`fract f1, f2;` `fract f3 = f1* f2; // in biased/unbiased rounding mode`
`fract16 f1, f2;` `fract32 f3 = mult_fr1x32(f1, f2);`	`fract f1, f2;` `long fract f3 = (long fract)f1* (long fract)f2;`
`fract16 f1;` `fract16 f2 = abs_fr1x16(f1);`	`fract f1;` `fract f2 = absr(f1);`
`fract16 f1;` `fract16 f2 = negate_fr1x16(f1);`	`fract f1;` `fract f2 = -f1;`
`fract16 f1;` `int n = norm_fr1x16(f1);`	`fract f1;` `int n = countlsr(f1);`
`fract32 f1, f2;` `fract32 f3 = add_fr1x32(f1, f2);`	`long fract f1, f2;` `long fract f3 = f1+ f2;`
`fract32 f1, f2;` `fract32 f3 = sub_fr1x32(f1, f2);`	`long fract f1, f2;` `long fract f3 = f1- f2;`
`fract32 f1;` `fract32 f2 = negate_fr1x32(f1);`	`long fract f1;` `long fract f2 = -f1;`
`fract32 f1;` `int n = norm_fr1x32(f1);`	`long fract f1;` `int n = countlslr(f1);`
`fract32 f1;` `fract16 = trunc_fr1x32(f1);`	`long fract f1;` `fract f2 = f1; // in truncation rounding mode`

`#include <fract2float_conv.h>`
`fract16 f1;`
`fract32 f2;`
`float f3;`
`f2 = fr16_to_fr32(f1);`
`f1 = fr32_to_fr16(f2);`
`f3 = fr16_to_float(f1);`
`f3 = fr32_to_float(f2); f1 = float_to_fr16(f3);`
`f2 = float_to_fr32(f3);`
	`fract f1;`
	`long fract f2;`
	`float f3; f2 = f1;`
	`f1 = f2;`
	`f3 = f1;`
	`f3 = f2;`
	`f1 = f3;`
	`f2 = f3;`

For convenience, built-in functions are also provided giving the same functionality on native fixed-point types, and it is simply necessary to change the built-in name replacing `fr` with `fx`. For example, if your original code says

```
#include <fract.h>
#include <builtins.h>
fract16 offset = 0.5r16;
```

```
fract16 add_offset(fract16 f) {
    return add_fr1x16(f, offset);
}
```

you can change it to:

```
#include <stdfix.h>
#include <builtins.h>

fract offset = 0.5r;
fract add_offset(fract f) {
    return add_fx1x16(f, offset);
}
```

although it is clearer to write:

```
#include <stdfix.h>
fract offset = 0.5r;

fract add_offset(fract f) {
    return f + offset;
}
```

There are a number of built-ins that do not map directly onto fixed-point arithmetic but similar functionality is available. See the ***fract16 and fract32 Built-In Functions and Native Fixed-Point Arithmetic with Similar Semantics*** table for details. These built-ins perform 1.31 fractional multiplication, rounding the result. However, the result may not be bit-identical to the result of native long fract multiplication, even in round-to-nearest mode, as the rounding performed by the native types is more exact than that provided by the built-ins. It is recommended that you use the native fixed-point arithmetic unless you require bit-exact results with respect to your previous implementation. In that case, you can use the bit-exact equivalent built-in functions, `mult_fx1x32x32`, `mult_fx1x32x32NS`, and `multr_fx1x32x32`.

Table 2-21: fract16 and fract32 Built-In Functions and Native Fixed-Point Arithmetic with Similar Semantics

fract16 or fract32 Built-In Function	*Native Fixed-Point Type Arithmetic*

`fract32 f1, f2;`
`fract32 f3 = mult_fr1x32x32(f1, f2);`
	`long fract f1, f2;`
	`long fract f3 = f1* f2 // in biased/`
	`unbiased rounding mode;`

`fract32 f1, f2;`
`fract32 f3 = multr_fr1x32x32(f1, f2);`
	`long fract f1, f2;`
	`long fract f3 = f1* f2 // in biased/`
	`unbiased rounding mode;`

`fract32 f1, f2;`
`fract32 f3 = mult_fr1x32x32NS(f1, f2);`
	`long fract f1, f2;`
	`long fract f3 = f1* f2 // in biased/`
	`unbiased rounding mode;`

There are many library functions that use `fract16` and `fract32` types. As a general rule, you can simply replace the `fr` with `fx` to obtain a library function that accepts and/or returns native fixed-point types instead. However, there is no fixed-point version of the vector type `fract2x16` or the complex fractional types

`complex_fract16` and `complex_fract32`, so special care must be taken when a mixture of native fixed-point types and vector or complex fractional types is used. The `fract2x16`, `complex_fract16`, and `complex_fract32` types can be used with the native fixed-point types so long as care is taken to access the data members with the constructor and accessor functions given in the *Constructor and Accessor Functions for Using Native Fixed-Point Types with Complex and Vector Fractional Types* table.

Table 2-22: Constructor and Accessor Functions for Using Native Fixed-Point Types with Complex and Vector Fractional Types

Built-In Function	*Description*
`complex_fract16 ccompose_fx_fr16(fract real, fract imag);`	Create a `complex_fract16` value from `fract`-typed real and imaginary parts.
`fract real_fx_fr16(complex_fract16 c);`	Extract the `fract`-typed real part of a `complex_fract16` value.
`fract imag_fx_fr16(complex_fract16 c);`	Extract the `fract`-typed imaginary part of a `complex_fract16` value.
`complex_fract32 ccompose_fx_fr32(long fract real, long fract imag);`	Create a `complex_fract32` value from `long fract`-typed real and imaginary parts.
`long fract real_fx_fr32(complex_fract32 c);`	Extract the `long fract`-typed real part of a `complex_fract32` value.
`long fract imag_fx_fr32(complex_fract32 c);`	Extract the `long fract`-typed imaginary part of a `complex_fract32` value.
`fract2x16 compose_fx_fr2x16(fract x, fract y);`	Create a `fract2x16` value from two `fract`-typed parts.
`fract low_of_fx_fr2x16(fract2x16 vec);`	Extract the `fract`-typed low part of a `fract2x16` value.
`fract high_of_fx_fx2x16(fract2x16 vec);`	Extract the `fract`-typed high part of a `fract2x16` value.

The naming convention for library functions that take a mixture of fixed-point type and `fract2x16`, `complex_fract16`, or `complex_fract32` types is to add `fx_` before the `fr2x16`, `fr16`, or `fr32` in the function name. You can check the name to use by consulting the documentation page for the library function. Note that function names that do not use `fract16` or `fract32` types do not need to be changed.

Fixed-Point Type Example

This section examines an example program to compute the variance of an array of 16-bit fractional values.

The variance of an array of values `samples[]` is given by:

$$variance = \frac{n \sum_{i=0}^{n-1} samples_i^2 - \left(\sum_{i=0}^{n-1} samples_i \right)^2}{n(n-1)}$$

where n is the number of samples in the array.

How does this map onto the fixed-point types? `samples` is an array of `fract` values, so in order to compute the sum of all the samples values, a type with greater range than a fractional type is needed. If there are fewer than 256 samples, it is certain that the sum will fit in an `accum` type without saturation occurring. The same argument applies to the sum of the squares of the `samples` elements.

However, the formula above also needs to calculate the intermediate result `sample_length * sum(samples[i] * samples[i])`. The multiplication by `sample_length` means that it is not certain that the result of the multiplication is within the range of an `accum` type.

An equivalent formula for the variance is:

$$variance = \frac{\displaystyle\sum_{i=0}^{n-1} samples_i^2 - \frac{\left(\displaystyle\sum_{i=0}^{n-1} samples_i\right)^2}{n}}{(n-1)}$$

This alternative definition means that the necessary intermediate values can be computed in an `accum` type. A possible implementation is given in the following example.

A Function to Compute the Variance of an Array of 16-bit Fractional Values

```
#include <stdfix.h>
#include <builtins.h>

// FX_CONTRACT ON ensures that the compiler recognizes
// accum += fract * fract idioms
#pragma FX_CONTRACT ON
fract fract_variance(const fract *samples, int sample_length) {
   fract variance = 0.0r;

   if (sample_length > 1) {
       #pragma FX_ROUNDING_MODE UNBIASED
       int i, saved_rnd_mod = set_rnd_mod_unbiased();
       accum diff, sum_of_samples = 0.0k, sum_of_squares = 0.0k;
       long fract mean;

       // this is guaranteed not to saturate
       // so long as sample_length = 255
       for (i = 0; i  sample_length; i++) {
           sum_of_samples += samples[i];
           sum_of_squares += samples[i] * samples[i];
       }
     mean = sum_of_samples / sample_length;
     diff = sum_of_squares - (mean * sum_of_samples);
     variance = diff / (sample_length - 1);
     restore_rnd_mod(saved_rnd_mod);
   }
```

```
    return variance;
}
```

Firstly, `stdfix.h` has been included in order to be able to use the natural spellings `fract` and `accum`. The next thing you might notice is the explicit use of `#pragma FX_CONTRACT ON`. Since this is the default setting of the `FX_CONTRACT` mode, this statement is not strictly necessary, but it is useful to document the assumptions made by the program.

It only makes sense to compute the variance if there is more than one sample, otherwise the function returns zero.

Next, the function sets the rounding mode. Here, unbiased rounding has been used to maintain the highest accuracy in the result. This is done by using the `FX_ROUNDING_MODE UNBIASED` pragma and `set_rnd_mod_unbiased` built-in function together, as discussed in Setting the Rounding Mode.

The loop computes the sum of the `samples` and the sum of the squares. Since `FX_CONTRACT` mode is `ON`, no precision is lost as the `fracts` are multiplied together and summed into the `accum` type.

After the loop, the sum of the `samples` is divided by the `sample_length` to give the mean sample value. This must be in the range [-1.0,1.0). It is stored into a `long fract` to retain as much accuracy as possible.

Next, the function computes the difference between the sum of the squares and the product of the mean and the sum of the `samples`. Since the absolute value of the mean is less than or equal to one, this product fits in an `accum` and, since this product and the sum of the squares are both non-negative, the difference must also fit in an `accum`.

Finally, the variance is computed by dividing this difference by one less than the `sample_length`. In theory, this value may be greater than one; in this case the returned value will be saturated to give `FRACT_MAX`.

Language Standards Compliance

The compiler supports code that adheres to the ISO/IEC 9899:1990 C standard, ISO/IEC 9899:1999 C standard, the ISO/IEC 14882:2003 C++ standard, and the ISO/IEC 14882:2011 C++ standard.

- C Mode
- C++ Mode

The compiler's level of conformance to the applicable ISO/IEC standards is validated using commercial test-suites from Plum Hall, Perennial, and Dinkumware.

C Mode

The compiler shall compile any program that adheres to a hosted implementation of the ISO/IEC 9899:1990 C standard, but it does not prohibit the use of language extensions (C/C++ Compiler Language Extensions) that are compatible with the correct translation of standard-conforming programs. To enable this mode, the `-c89` switch should be used; see -c89.

The compiler shall compile any program that adheres to a freestanding implementation of the ISO/IEC 9899:1999 C standard, but it does not prohibit the use of language extensions (C/C++ Compiler Language Extensions) that are

compatible with the correct translation of standard-conforming programs. The compiler does supports the C99 keyword _Complex, but not _Imaginary. The ISO/IEC 9899:1990 C standard library provided in C89 mode is used in C99 mode. This is the default mode; see -c99.

In C mode, the best standard conformance is achieved using the default switches and the following non-default switches:

- -double-size-{32 | 64} (see also Floating-Point Data Size)
- -full-io
- -decls-{weak|strong}
- Enumeration Type Implementation Details

The floating-point arithmetic emulation library used by the compiler is based on IEEE-754; see IEEE Floating-Point Implementation for deviations.

The language extensions cannot be disabled to ensure strict compliance to the language standards. However, when compiling for MISRA-C (MISRA-C Compiler Overview) compliance checking, language extensions are disabled.

When the -c89 switch is enabled, these extensions already include many of the ISO/IEC 9899:1999 standard features. The following features are only available in C99 mode.

- Type qualifiers may appear more than once in the same specifier-qualifier-list.
- __func__ predefined identifier is supported.
- Universal character names (\u and \U) are accepted.
- The use of function declarations with non-prototyped parameter lists are faulted.
- The first statement of a for-loop can be a declaration, not just restricted to an expression.
- Type qualifiers and static are allowed in parameter array declarators.

C++ Mode

The compiler shall compile any program that adheres to a hosted implementation of the ISO/IEC 14882:2003 C++ standard, but it does not prohibit the use of language extensions (C/C++ Compiler Language Extensions) that are compatible with the correct translation of standard-conforming programs. A library fully conforming to the ISO/IEC 14882:2003 C++ standard is available (-full-cpplib), but by default the Abridged Library is used, which is a proper subset of the full Standard C++ Library and is designed specifically for the needs of the embedded market.

Using the -c++11 switch, support for many of the ISO/IEC 14882:2011 C++ standard language features can be enabled. However, there is no support for library features that are new to ISO/IEC 14882:2011. The key language features supported by the -c++11 switch are (please refer to the ISO/IEC 14882:2011 standard for more detail on usage):

- The static_assert construct
- friend class syntax is extended to all nonclass types

- Mixed string literal concatenations

- The C99-style `_Pragma` operator

- An explicit instantiation maybe prefixed with the `extern` keyword

- The keyword `auto` can be used as a type specifier

- The keyword `decltype`

- Scoped enumeration types (defined with the keyword sequence `enum class`)

- Lambdas

- Rvalue references

- Ref-qualifiers

- Functions can be "`deleted`"

- Special member functions can be explicitly "`defaulted`"

- Move constructors and move assignment operators

- Conversion functions can be marked `explicit`

- The keyword `nullptr` can be used as both a null pointer constant and a null pointer-to-member constant

- The context-sensitive keyword `final`

- Alias and alias template declarations

- Variadic templates

- U-literals and the `char16_t` and `char32_t` keywords

- Inline namespaces

- Initializer lists

- The `noexcept` specifier and keyword

- Range-based "`for`" loops

- Non-static data member initializers

- `constexpr` keyword

- Unrestricted unions

- Delegating constructors

- Ref-qualifiers on member functions

- Raw strings

- UTF-8 strings

In C++ mode, the best possible standard conformance is achieved using the following default switches:

- -no-anach

- -no-friend-injection

- -no-implicit-inclusion

- -std-templates

In addition, the best possible standard conformance is achieved using the following non-default switches:

- -double-size-{32 | 64}

- -eh

- -full-cpplib

- -full-io

- -decls-{weak|strong}

- -rtti

MISRA-C Compiler

This section provides an overview of MISRA-C compiler and MISRA-C:2004 Guidelines.

MISRA-C Compiler Overview

The Motor Industry Software Reliability Association (MISRA) in 1998 published a set of guidelines for the C Programming Language to promote best practice in developing safety related electronic systems in road vehicles and other embedded systems. The latest release of MISRA-C:2004 has addressed many issues raised in the original guidelines specified in MISRA-C:1998. Complex rules are now split into component parts. There are 121 mandatory rules and 20 advisory rules. The compiler issues a discretionary error for mandatory rules and a warning for advisory rules. More information on MISRA-C can be obtained at `http://www.misra.org.uk/`.

The compiler detects violations of the MISRA rules at compile-time, link-time, and run-time. It has full support for the MISRA-C:2004 Guidelines, including the Technical clarifications given by MISRA-C:2004 Technical Corrigendum 1. The majority of MISRA rules are easy to interpret. Those that require further explanation can be found in Rule Descriptions. As a documented extension, the compiler supports the integral types `long long` and `unsigned long long`. No other language extensions are supported when MISRA checking is enabled. Common extensions, such as the keywords `section` and `inline`, are not allowed in the MISRA mode, but the same effects can be achieved by using #pragma section/#pragma default_section and #pragma inline. Rules can be suppressed by the use of command-line switches or the

NOTE: The run-time checking that is used for validating a number of rules should not be used in production code. The cost of detecting these violations is expensive in both run-time performance and code size. A

subset of these run-time checks can also be enabled when MISRA-C is not enabled. For more information, see Run-Time Checking.

Refer to the *C Mode (MISRA) Compiler Switches* table in C/C++ Compiler Command-Line Switches for the list of MISRA-C command-line switches.

MISRA-C Compliance

The MISRA-C:2004 Guidelines document is an essential reference for ensuring that code developed or requiring modification complies to these Guidelines. A rigorous checking tool, such as this compiler, makes achieving compliance a lot easier than using a less capable tool or simply relying on manual reviews of the code. The MISRA-C:2004 Guidelines document describes a compliance matrix that a developer uses to ensure that each rule has a method of detecting the rule violation. A compliance checking tool is a vital component in detecting rule violations. It is recognized in the Guidelines document that in some circumstances it may be necessary to deviate from the given rules. A formal procedure has to be used to authorize these deviations rather than an individual programmer having to deviate at will.

Using the Compiler to Achieve Compliance

The CCES compiler is one of the most comprehensive MISRA-C:2004 compliance checking tools available. The compiler provides command-line switches (C Mode (MISRA) Compiler Switch Descriptions) and diagnostic control pragmas (Diagnostic Control Pragmas) to enable you to achieve MISRA-C:2004 compliance.

During development it is recommended that the application is built with maximum compliance enabled.

Use the `-misra-strict` command-line switch to detect the maximum number of rule violations at compile-time. However, if existing code is being modified, using `-misra-strict` may result in a lot of errors and warnings. The majority are usually common rule violations that are mainly advisory and typically found in header files as a result of macro expansion. These can be suppressed using the `-misra` command-line switch. This has the potential benefit of focusing change on individual source file violations, before changing headers that may be shared by more than one project.

The `-misra-no-cross-module` command-line switch disables checking rule violations that occur across source modules. During development some external variables may not be fully utilized and rather than add in artificial uses to avoid rule violations, use this switch.

The `-misra-no-runtime` command-line switch disables the additional run-time overheads imposed by some rules. During development these checks are essential in ensuring code executes as expected. Use this switch in release mode to disable the run-time overheads.

You can use the `-misra-testing` command-line switch during development to record the behavior of executable code. Although the MISRA-C:2004 Guidelines do not allow library functions such as those as defined in the header `<stdio.h>`, it is recognized that they are an essential part of validating the development process.

During development, it is likely that you will encounter areas where some rule violations are unavoidable. In such circumstances you should follow the procedure regarding rule deviations described in the MISRA-C:2004 Guidelines document. Use the `-Wmis_suppress` and `-Wmis_warn` switches to control the detection of rule violations for whole source files.

Finer control is provided by the diagnostic control pragmas. These pragmas allow you to suppress the detection of specified rule violations for any number of C statements and declarations.

Example

```
#include <misra_types.h>
#include <defBF532.h>
#include "proto.h"     /* prototype for func_state and my_state */

int32_t func_state(int32_t state)
{
    return state & TIMOD;
            /* both operands signed, violates rule 12.7 */
}

#define my_flag 1

int32_t my_state(int32_t state)
{
   return state & my_flag;
            /* both operands signed, violates rule 12.7 */
}
```

In the above example, `<defBF532.h>` uses signed masks and signed literal values for register values. The code is meaningful and trusted in this context. You may suppress this rule and document the deviation in the code. For code violating the rule that is not from the system header, you may wish to rewrite the code:

```
#include <misra_types.h>
#include <defBF532.h>
#include "proto.h"     /* prototype for func_state and my_state */

#ifdef _MISRA_RULES
#pragma diag(push)
#pragma diag(suppress:misra_rule_12_7:
                  "Using the def file is a safe and justified
                   deviation for rule 12.7")
#endif /* _MISRA_RULES */

int32_t func_state(int32_t state)
{
   return state & TIMOD;
                /* both operands signed, violates rule 12.7 */
}

#ifdef _MISRA_RULES
#pragma diag(pop)
                /* allow violations of 12.7 to be detected again */
#endif /* _MISRA_RULES  */

#define my_flag 1u
uint32_t my_state(uint32_t state)
```

```
{
    return state & my_flag;    /* o.k both unsigned */
}
```

Rule Descriptions

The following are brief explanations of how some of the MISRA-C rules are supported and interpreted in this CCES release due to the fact that some rules are handled in a nonstandard way, or some are not handled at all:

- *Rule 1.4 (required): The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.*

  The compiler and linker fully support this requirement.

- *Rule 1.5 (required): Floating-point implementations should comply with a defined floating-point standard.*

  Refer to Floating-Point Binary Formats.

- *Rule 2.4 (advisory): Sections of code should not be "commented out".*

  A diagnostic is reported if one of the following is encountered inside of a comment.
  ```
  - character `{` or `}'
  - character `;' followed by a new-line character
  ```

- *Rule 5.1 (required): Identifiers (internal and external) shall not rely on the significance of more than 31 characters.*

  This rule is only enforced when the `-misra-strict` compiler switch is enabled (-misra-strict).

- *Rule 5.5 (advisory): No object or function identifier with static storage duration should be reused.*

  This rule is enforced by the compiler prelinker. The compiler generates `.misra` extension files that the prelinker uses to ensure that the same identifier is not used at file-scope within another module. This rule is not enforced if the `-misra-no-cross-module` compiler switch is specified (-misra-no-cross-module).

- *Rule 5.7 (advisory): No identifier shall be reused.*

  This rule is limited to a single source file. The rule is only enforced when the `-misra-strict` compiler switch is enabled (-misra-strict).

- *Rule 6.3 (advisory): typedefs that indicate size and signedness should be used in place of basic types.*

  The `typedefs` for the basic types are provided by the system header files `<stdint.h>` and `<stdbool.h>`. The rule is only enforced when the `-misra-strict` compiler switch is enabled (-misra-strict).

- *Rule 6.4 (advisory): Bit fields shall only be defined to be of type unsigned int or signed int.*

  The rule regarding the use of plain `int` is only enforced when the `-misra-strict` compiler switch is enabled (-misra-strict).

- *Rule 8.1 (required): Functions shall have prototype declarations and the prototype shall be visible at both the function definition and the call.*

  For static and inline functions, this rule is only enforced when the `-misra-strict` compiler switch is enabled (-misra-strict).

- *Rule 8.2 (required): Whenever an object or function is declared or defined, its type shall be explicitly stated.*

  For function `main`, this rule is only enforced when the `-misra-strict` switch is enabled.

- *Rule 8.5 (required): There shall be no definitions of objects or functions in a header file.*

  This rule is only enforced when the `-misra-strict` switch is enabled (-misra-strict).

- *Rule 8.8 (required): An external object or function shall be declared in one and only one file.*

  This rule is enforced by the compiler prelinker. The compiler generates `.misra` extension files that the prelinker uses to ensure that the global is used in another file. The rule is not enforced if the `-misra-no-cross-module` switch is enabled (-misra-no-cross-module).

- *Rule 8.10 (required): All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.*

  This rule is enforced by the compiler prelinker. The compiler generates `.misra` extension files that the prelinker uses to ensure that the global is used in another file. The rule is not enforced if the `-misra-no-cross-module` switch is enabled (-misra-no-cross-module).

- *Rule 9.1 (required): All automatic variables shall have been assigned a value before being used.*

  The compiler attempts to detect some instances of violations of this rule at compile-time. There is additional code added at run-time to detect unassigned scalar variables. The additional integral types with a size less than an int are not checked by the additional run-time code. This check is also available separately, via the `-rtcheck` switch (-rtcheck) and the `-rtcheck-unassigned` switch (-rtcheck-unassigned). The run-time code is not added if the `-misra-no-runtime` compiler switch is enabled (-misra-no-runtime), or if the `-no-rtcheck-unassigned` switch is enabled (-no-rtcheck-unassigned).

- *Rule 10.5 (required): If the bitwise operators ~ and are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.*

  When constant-expressions violate this rule, they are only detected when the `-misra-strict` compiler switch is enabled (-misra-strict).

- *Rule 11.3 (advisory): A cast shall not be performed between a pointer type and an integral type.*

  The compiler always allows a constant of integral type to be cast to a pointer to a volatile type.
  ```
  volatile int32_t *n;
  n = (volatile int32_t *)10;
  ```

  There is only one case where this rule is not applied.

```
int32_t *n;
n = (int32_t *)10;
```

- *Rule 12.4 (required): The right-hand operand of a logical && or || operator shall not contain side-effects.*

  A function call used as the right-hand operand will not be faulted if it is declared with an associated `#pragma pure` directive.

- *Rule 12.7 (required): Bitwise operators shall not be applied to operands whose underlying type is signed.*

  The compiler will not enforce this rule if the two operands are constants.

- *Rule 12.8 (required): The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.*

  If the right-hand operand is not a constant expression, the violation will be checked by additional run-time code when `-misra-no-runtime` is not enabled. If both operands are constants, the rule is only enforced when neither the `-misra-strict` compiler switch (-misra-strict) nor the `-no-rtcheck-shift-check` switch (-no-rtcheck-shift-check) are enabled. This check is also available separately, via the `-rtcheck` switch (-rtcheck), and the `-rtcheck-shift-check` switch (-rtcheck-shift-check).

- *Rule 12.12 (required): The underlying bit representations of floating-point values shall not be used.*

  MISRA-C rules such as 11.4 prevent casting of bit-patterns to floating-point values. Hexadecimal floating-point constants are also not allowed when MISRA-C switches are enabled.

- *Rule 13.2 (advisory): Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.*

  The compiler treats variables which use the type `bool` (a typedef is declared in `<stdbool.h>`) as "effectively Boolean" and will not raise an error when these are implicitly tested as zero, as follows:

```
bool b = 1;
    if (bool)
        ...;
```

- *Rule 13.7 (required): Boolean operations whose results are invariant shall not be used.*

  The compiler does not detect cases where there is a reliance on more than one conditional statement. Constant expressions violating the rule are only detected when the `-misra-strict` compiler switch is enabled (-misra-strict).

- *Rule 16.2 (required): Functions shall not call themselves, either directly or indirectly.* A compile-time check is performed for a single file. Run-time code is added to ensure that functions do not call themselves directly or indirectly, but this code is not generated if the `-misra-no-runtime` compiler switch is enabled (-misra-no-runtime).

- *Rule 16.4 (required): The identifiers used in the declaration and definition of a function shall be identical.*

A declaration of a parameter name may have one leading underscore that the definition does not contain. This is to prevent name clashing. If the `-misra-strict` compiler switch is enabled (-misra-strict), the underscore is significant and results in the violation of this rule.

- *Rule 16.5 (required): Functions with no parameters shall be declared and defined with the parameter list void.*

  Function `main` shall only be reported as violating this rule if the `-misra-strict` compiler switch is enabled (-misra-strict).

- *Rule 16.10 (required): If a function returns error information, then the error information shall be tested.*

  A function declared with return type `bool`, which is a `typedef` declared in header file `<stdbool.h>` will be faulted if the result of the call is not used.

- *Rule 17.1 (required): Pointer arithmetic shall only be applied to pointers that address an array or array element.*Checking is performed at run-time. A run-`time` function looks at the value of the pointer and checks to see whether it violates this rule. This check is also available via the `-rtcheck` (-rtcheck) and `-rtcheck-arr-bnd` (-rtcheck-arr-bnd) switches. It can be disabled via the `-no-rtcheck-arr-bnd` switch (-no-rtcheck-arr-bnd).

- *Rule 17.2 (required): Pointer subtraction shall only be applied to pointers that address elements of the same array.*

  Checking is performed at runtime. A run-`time` function looks at the value of the pointers and checks to see whether it violates this rule.

- *Rule 17.3 (required): >, >=, <, <= shall not be applied to pointers that address elements of different arrays.*

  Checking is performed at run-time. A run-`time` function looks at the value of the pointers and checks to see whether it violates this rule.

- *Rule 17.6 (required): The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.*

  Rule is not enforced under the following circumstances: if the address of a local variable is passed as a parameter to another function, the compiler cannot detect whether that address has been assigned to a global object.

- *Rule 18.2 (required): An object shall not be assigned to an overlapping object.*

  The rule is not enforced by the compiler.

- *Rule 18.3 (required): An area of memory shall not be reused for unrelated purposes.*

  The rule is not enforced by the compiler.

- *Rule 19.7 (advisory): A function shall be used in preference to a function-like macro.*

  The rule is only enforced when the compiler option `-misra-strict` is enabled.

- *Rule 19.15 (required): Precautions shall be taken in order to prevent the contents of a header file being included twice.*

The compiler will report this violation if a header file is included more than once and does not prevent re-declaration of types, variables, or functions.

- *Rule 20.3 (required): The validity of values passed to library functions shall be checked.*

  This is not enforced by the compiler. The rule puts the responsibility on the programmer.

- *Rule 20.4 (required): Dynamic heap memory allocation shall not be used.*

  Prototype declarations for functions performing heap allocation should be declared with an associated `#pragma misra_func(heap)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- *Rule 20.7 (required): The setjmp macro and longjmp function shall not be used.*

  Prototype declarations for these should be declared with an associated `#pragma misra_func(jmp)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- *Rule 20.8 (required): The signal handling facilities of <signal.h> shall not be used.*

  Prototype declarations for functions in this header should be declared with an associated `#pragma misra_func(handler)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- *Rule 20.9 (required): The input/output library <stdio.h> shall not be used.*

  Prototype declarations for functions in this header should be declared with an associated `#pragma misra_func(io)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- *Rule 20.10 (required): The library functions atof, atoi and atol from library <stdlib.h> shall not be used.*

  Prototype declarations for these functions should be declared with an associated `#pragma misra_func(string_conv)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- *Rule 20.11 (required): The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.*

  Prototype declarations for these functions should be declared with an associated `#pragma misra_func(system)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- *Rule 20.12 (required): The time handling functions of library <time.h> shall not be used.*

  Prototype declarations for these functions should be declared with an associated `#pragma misra_func(time)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- *Rule 21.1 (required): Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.*

  The compiler performs some static checks on uses of unassigned variables before conditional code and use of constant expressions. The compiler performs run-time checks for arithmetic errors, such as division by zero, array bound errors, unassigned variable checking, and pointer dereferencing. Run-time checking has a negative effect on code performance. The -misra-no-runtime compiler switch turns off the run-time checking (-misra-no-runtime).

Run-Time Checking

The compiler provides support for detecting common programming mistakes, such as dereferencing a NULL pointer, or accessing an array beyond its bounds. The compiler does this by generating additional code to check for such conditions at runtime. Such code occupies space and incurs a performance penalty, so you should only use run-time checking when developing and debugging your application; products for release should always have run-time checking disabled.

The compiler's run-time checks are a subset of those enabled when MISRA-C run-time checking is active. For more information, see MISRA-C Compiler.

The following sections describe run-time checking in detail.

- Enabling Run-Time Checking

- Supported Run-Time Checks

- Response When Problems Are Detected

- Limitations of Run-Time Checking

Enabling Run-Time Checking

Because of the associated overheads, run-time checking is disabled by default. You can enable run-time checking:

- By specifying command-line switches.

- Through the IDE, via run-time checking options under *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Run-time Checks*.

In both cases, you can enable all supported run-time checks, or just enable specific subsets.

Once run-time checking is enabled to some level, you can further turn that checking off and on again within your code, with pragmas. This allows you to narrow your focus down to particular functions, or to exclude certain functions from checking.

Command-Line Switches for Run-Time Checking

The following switches turn run-time checking on:

- `-rtcheck` - turns on all run-time checks.
- `-rtcheck-arr-bnd` - turns on checking of array boundaries.
- `-rtcheck-div-zero` - turns on checking for division by zero.
- `-rtcheck-heap` - turns on checking of heap operations.
- `-rtcheck-null-ptr` - turns on checking for NULL pointer dereferencing.
- `-rtcheck-shift-check` - turns on checking of shift operations.
- `-rtcheck-stack` - turns on checking for stack overflow.
- `-rtcheck-unassigned` - turns on checking for use of variables before they've been assigned values.

The following switches are used to turn run-time checking off:

- `-no-rtcheck` - turns off all run-time checks.
- `-no-rtcheck-arr-bnd` - turns off checking of array boundaries.
- `-no-rtcheck-div-zero` - turns off checking for division by zero.
- `-no-rtcheck-heap` - turns off checking of heap operations.
- `-no-rtcheck-null-ptr` - turns off checking for NULL pointer dereferencing.
- `-no-rtcheck-shift-check` - turns off checking of shift operations.
- `-no-rtcheck-stack` - turns off checking for stack overflow.
- `-no-rtcheck-unassigned` - turns off checking for use of variables before they've been assigned values.

You can use combinations of these switches to enable the subset you require. For example, the following two sets of switches are equivalent:

- `-rtcheck -no-rtcheck-arr-bnd -no-rtcheck-div-zero \ -no-rtcheck-heap -no-rtcheck-stack`
- `-rtcheck-null-ptr -rtcheck-shift-check -rtcheck-unassigned`

For more information, see -rtcheck.

Pragmas for Run-Time Checking

The following pragmas are used to enable and disable run-time checks.

- `#pragma rtcheck(on)` - rurns on that subset of run-time checking that has been enabled by command-line switches.
- `#pragma rtcheck(off)` - turns off all run-time checking.

Note that these pragmas do not affect which run-time checks apply-use command-line switches to select the appropriate checks, then use the pragmas to enable those checks during compilation of your functions of interest.

For more information, see Run-Time Checking Pragmas.

Supported Run-Time Checks

The following run-time checks are supported by the compiler:

Array Boundary Checks

When generating code to access arrays, the compiler generates additional code to see whether the location accessed falls within the boundaries of a live automatic array.

Division by Zero Checks

When generating code to perform an integer or floating-point division, the compiler generates additional code to check that the divisor is non-zero.

Heap Checks

The debugging version of the heap library checks for leaks, multiple frees of the same pointer, writes beyond the bounds of an allocation, and so on.

NULL Pointer Checks

When generating code to read the value pointed to by a pointer, the compiler generates additional code to verify that the pointer is not NULL.

Shift Checks

When generating code to shift a value X by some amount Y, the compiler generates additional code to check that:

- Y is not a negative value.
- Y is less than the number of bits required to represent X's type.

Stack Overflow Checks

When increasing the amount of stack space in use, the compiler generates additional code to verify that the bounds of the current stack are not about to be exceeded.

Unassigned Variable Checks

When generating code to read the value of a variable, the compiler generates additional code to make sure a value has previously been assigned to the variable.

Response When Problems Are Detected

In most cases, the additional code generated by the compiler includes code for emitting a diagnostic message to the `stderr` stream. This message is emitted when the run-time check finds a problem.

When stack overflow is detected, however, the generated code transfers control to the special label `_adi_stack_overflowed`, as emitting a diagnostic to the stderr stream would require additional stack space. The IDE normally places a breakpoint on the `_adi_stack_overflowed` label. For more information, see Stack Overflow Detection in the *Optimal Performance from C/C++ Source Code* chapter.

The heap debugging library also provides support for logging problems to a file instead of reporting them immediately to the stderr stream. For more information, see Heap Debugging in the *Achieving Optimal Performance From C/C++ Source Code* chapter.

Limitations of Run-Time Checking

Besides the space/performance overheads incurred by the additional code, the following limitations apply to run-time checking:

- Compiled code only: Because the run-time checks rely on additional code emitted during function compilation, the run-time checks can only apply to code compiled by the compiler, while run-time checks are enabled. Hand-written assembly or previously-compiled code cannot make benefit from run-time checking.

- No asm statements: The compiler has no visibility into the contents of asm statements, so any actions carried out by asm statements will not be checked by any enabled run-time checking. or more information, see Inline Assembly Language Support Keyword (asm).

- Stdio support required: Because the generated diagnostics are emitted to the stderr stream, run-time checking is only beneficial when the application supports the standard error stream, and the stream is attached to some suitable output device (such as the IDE console, which is the usual case when running an application within the debugger).

C/C++ Compiler Language Extensions

The compiler supports extensions to the ANSI/ISO standard for the C and C++ languages. These extensions add support for DSP hardware and permit some C++ programming features when compiling in C mode. Many of the extensions are also available when compiling in C++ mode. Additional extensions for compatibility with GNU C++ can be enabled with the -g++ switch.

This section contains information on ISO/IEC 9899:1999 standard features that are supported in C89 mode:

- Function Inlining

- Variable Argument Macros

- Restricted Pointers

- Variable-Length Arrays

- Non-Constant Initializer Support

- Designated Initializers

- Hexadecimal Floating-Point Numbers

- Declarations Mixed With Code

- Compound Literals Support

- C++ Style Comments

- Enumeration Constants That Are Not int Type

- Boolean Type Support Keywords (bool, true, false)

This section also contains information on other language extensions:

- Native Fixed-Point Types fract and accum

- Inline Assembly Language Support Keyword (asm)

- Memory Banks

- Placement Support Keyword (section)

- Placement of Compiler-Generated Code and Data

- Long Identifiers

- Compiler Built-In Functions

- Pragmas

- Preprocessor-Generated Warnings

The additional keywords that are part of the C/C++ extensions do not conflict with ANSI C/C++ keywords. The formal definitions of these extension keywords are prefixed with a leading double underscore (__). Unless the -no-extra-keywords command-line switch is used, the compiler defines the shorter form of the keyword extension that omits the leading underscores. For more information, see the brief descriptions of each switch in C/C++ Compiler Common Switch Descriptions.

You might exclusively use the longer form (such as __inline) if porting a program that uses the extra Analog Devices keywords as identifiers. For example, if a program declares local variables, such as asm or inline, use the -no-extra-keywords switch. If you need to declare a function as inline, use __inline.

The *Keyword Extensions* table and the *Operational Extensions* table provide descriptions of each extension and direct you to sections that describe each extension in more detail.

Table 2-23: Keyword Extensions

Keyword Extensions	Description
`inline`	Directs the compiler to integrate the function code into the code of its callers. For more information, see Function Inlining.
`asm()`	Places Blackfin core assembly language commands directly in your C/C++ program. For more information, see Inline Assembly Language Support Keyword (asm).
`bank("string")`	Specifies a name which the user assigns to associate declarations that reside in particular memory banks. For more information, see Memory Banks.
`section("string")`	Specifies the section in which an object or function is placed. For more information, see Placement Support Keyword (section).
`booltruefalse`	Specifies a Boolean type. For more information, see Boolean Type Support Keywords (bool, true, false).
`restrict`	Specifies restricted pointer features. For more information, see Restricted Pointers.

Table 2-24: Operational Extensions

Operational Extensions	Description
Non-constant initializers	Permits the use of non-constants as elements of aggregate initializers for automatic variables. For more information, see Non-Constant Initializer Support.
Designated initializers	Specifies elements of an aggregate initializer in arbitrary order. For more information, see Designated Initializers.
Variable-length arrays	Creates local arrays with a variable size. For more information, see Variable-Length Arrays.
Long identifiers	Supports identifiers of up to 1022 characters in length. For more information, see Long Identifiers.
Preprocessor-generated warnings	Generates warning messages from the preprocessor. For more information, see Preprocessor-Generated Warnings.
C++ style comments	Allows for `//` C++ style comments in C programs. For more information, see C++ Style Comments.

Function Inlining

The `inline` keyword directs the compiler to integrate the code for the function you declare as `inline` into the code of its callers. Inline function support and the `inline` keyword is a standard feature of the ISO/IEC 14882:2003 C++ standard and the ISO/IEC 9899:1999 C standard; the `ccblkfn` compiler provides this keyword as an extension when the `-c89` switch is enabled. For more information, see -c89.

Function inlining eliminates the function call overhead and increases the speed of your program's execution. Argument values that are constant and that have known values may permit simplifications at compile time so that not all of the inline function's code need be included.

The following example shows a function definition that uses the `inline` keyword.

```
inline int max3 (int a, int b, int c) {
    return max (a, max(b, c));
}
```

The compiler can decide not to inline a particular function declared with the `inline` keyword; a diagnostic remark of `cc1462` is issued if the compiler chooses to do this. The diagnostic can be raised to a warning by use of the `-Wwarn` switch. For more information, see -W{annotation|error|remark|suppress|warn} *number[, number...]*.

Function inlining can also occur by use of the `-Oa` (automatic function inlining) switch (-Oa which enables the inline expansion of C/C++ functions that are not necessarily declared inline in the source code. The amount of auto-inlining the compiler performs is controlled using the `-Ov` (optimize for speed versus size) switch.

The compiler follows a specific order of precedence when determining whether a call can be inlined. The order is:

1. If the definition of the function is not available (for example, it is a call to an external function), the compiler cannot inline the call.

2. If the `-never-inline` switch has been specified (-never-inline), the compiler does not inline the call. If the call is to a function that has `#pragma always_inline` specified (see Inline Control Pragmas) a warning also is issued

3. If the call is to a function that has `#pragma never_inline` specified, the call is not inlined.

4. If the call is via a pointer-to-function, the call will not be inlined unless the compiler can prove that the pointer will always point to the same function definition.

5. If the call is to a function that has a variable number of arguments, the call is not inlined.

6. If the module contains `asm` statements at global scope (outside function definitions), the call may not be inlined because the `asm` statement restricts the compiler's ability to reorder the resulting assembly output.

7. If the call is to a function that has `#pragma always_inline` specified, the call is inlined. If the call exceeds the current speed/space ratio limits, the compiler will issue a warning, but will still inline the call.

8. If the call is to a function that has the `inline` qualifier or `#pragma inline` specified, and the `-always-inline` switch has been specified, the compiler will inline the call. If the call exceeds the current speed/space ratio limits, the compiler will issue a warning, but will still inline the call.

9. If the caller and callee are mapped to different code sections, the call will not be inlined unless the callee has the `inline` qualifier or has `#pragma inline` specified.

10. If the call is to a function that has the `inline` qualifier or has `inline` specified, and optimization is enabled, the called function will be compared against the current speed/size ratio limits for code size and stack size. The calling function will also be examined against these limits. Depending on the limits and the relative sizes of the caller and callee, the inlining may be rejected.

11. If the call is to a function that does not have the `inline` qualifier or `#pragma inline`, and does not have `#pragma weak_entry`, then if the `-Oa` switch has been specified to enable automatic inlining, the called function will be considered as a possible candidate for inlining, according to the current speed/size ratio limits, as if the `inline` qualifier were present.

The compiler bases its code-related speed/size comparisons on the −Ov switch (-Ov *num*). When −Ov is in the range 1...100, the compiler performs a calculation upon the size of the generated code using the −Ov value, and this will determine whether the generated code is "too large" for inlining to occur. When −Ov has the value 1, only very small functions are considered small enough to inline; when −Ov has the value 100, larger functions are more likely to be considered suitable as well.

When −Ov has the value 0, the compiler is optimizing for space. The speed/space calculation will only accept a call for inlining if it appears that the inlining is likely to result in less code than the call itself would (although this is an approximation, since the inlining process is a high-level optimization process, before actual machine instructions have been selected).

Inlining and Optimization

The inlining process operates regardless of whether optimization has been selected (although if optimization is not enabled, then inlining will only happen when forced by `#pragma always_inline` or the `-always-inline` switch). The speed/size calculation still has an effect, although an optimized function is likely to have a different size from a non-optimized one, which is smaller (and therefore more likely to be inlined) and dependent on the kind of optimization done.

A non-optimized function has loads and stores to temporary values which are optimized away in the optimized version, but an optimized function may have unrolled or vectorized loops with multiple variants, selected at run-time for the most efficient loop kernel. So an optimized function may run faster, but not be smaller.

Given that the optimization emphasis may be changed within a module - or even turned off completely - by the optimization pragmas, it is possible for either, both, or neither of the caller and callee to be optimized. The inlining process still operates, and is only affected by this in as far as the speed/size ratios of the resulting functions are concerned.

Inlining and Out-of-Line Copies

In C++ mode, the compiler conforms to the ISO/IEC:14882:2003 C++ standard and the ISO/IEC:14882:2011 C++ standard. In this mode the compiler will generate an out-of-line copy if the address of the function has been taken or the compiler has not been able to inline the call. Also, the out-of-line copy will have external linkage and be generated once by the prelinker if the function has been declared `extern inline`.

The following paragraphs describe the differences between C89 mode(-c89) and C99 mode(-c99). The use of inline is standard conforming in C99 mode(-c99) and is an extension in C89 mode(-c89).

An inline function declared `static` behaves the same whether it is declared in C89 mode(-c89) or C99 mode (-c99). No out-of-line copy is generated if all calls to the function are inlined and the address of the function is not taken.

An inline function declared with no storage class specifier behaves as follows. In C99 mode(-c99), if the address of the function is taken, then all calls not inlined within the current translation refer to an externally defined instance of the function, not the instance declared within this translation unit. In C99 mode(-c99), if the address of the function is not taken, it behaves as if the function has been declared with the keyword `static`. In C89 mode(-c89) it always behaves as if the function has been declared with the keyword `static`.

An inline function declared `extern` behaves differently in C99 mode(-c99) and C89 mode(-c89). In C99 mode(-c99) an external definition is always created. Therefore such a declaration should not be within a header file as this will result in a multiply defined symbol. In C89 mode (-c89) no out-of-line copy is created; therefore, if the address of the function is taken or a call is not inlined then an external reference is created and must be satisfied elsewhere.

Inlining and Global asm Statements

Inlining imposes a particular ordering on functions. If functions A and B are marked as inline, and each calls the other, only one of the `inline` qualifiers can be followed. Depending on which the compiler chooses to apply, either A will be generated with inline versions of B, or B will be generated with inline versions of A. Either case may result in no out-of-line copy of the inlined function being generated. The compiler reorders the functions within a module to get the best inlining result. Functionally, the code is the same, but this affects the resulting assembly file.

When global `asm` statements are used with the module, between the function definitions, the compiler cannot do this reordering process, because the `asm` statement might affect the behavior of the assembly code that is generated from the following C function definitions. Because of this, global `asm` statements can greatly reduce the compiler's ability to inline a function call.

Inlining and Sections

Before inlining, the compiler checks any section directives or pragmas on the function definitions. For example,

```
section("secA") inline int add(int a, int b) { return a + b; }
section("secB") int times_two(int a) { return add(a, a); }
```

Since `add()` and `times_two()` are to be generated into different code sections, this call is ignored during the inlining process, so the call is not inlined. If the callee is marked with `#pragma always_inline` (#pragma always_inline), however, or the `-always-inline` switch (-always-inline) is in force, the compiler will inline the call despite the mismatch in sections.

Inlining and Run-Time Checking

When run-time checking is enabled, the compiler generates the additional code for the checks when the function is first defined. The implications for function inlining are as follows:

- When a function defined with run-time checking enabled is inlined into a function without run-time checking enabled, the inlined version still includes the run-time checks.

- When a function defined with run-time checking disabled is inlined into a function with run-time checking enabled, the inlined version does not acquire any run-time checks.

For more information, see Run-Time Checking.

Variable Argument Macros

This ISO/IEC 9899:1999 C standard feature is enabled as an extension in C89 mode and in C++ mode. The final parameter in a macro declaration may be an ellipsis (`...`) to indicate the parameter stands for a variable number of arguments. In the replacement text for the macro, the predefined name `__VA_ARGS__` represents the parameters

that were supplied for the ellipsis in the macro invocation. At least one argument must be provided for the ellipsis, in an invocation.

For example,

```
#define  tracec99(file,line,...) logmsg(file,line, __VA_ARGS__)
```

can be used with differing numbers of arguments: the following statements:

```
tracec99("a.c", 999, "one", "two", "three");
tracec99("a.c", 999, "one", "two");
tracec99("a.c", 999, "one");
tracec99("a.c", 999);
```

expand to the following code:

```
logmsg("a.c", 999, "one", "two", "three");
logmsg("a.c", 999, "one",  "two");

logmsg("a.c", 999, "one");
logmsg("a.c", 999, ); // error - must provide an argument
```

NOTE: This variable argument macro syntax comes from the ISO/IEC 9899:1999 C standard. The compiler supports C99 variable argument macro formats in C89, C99, and C++ modes. In addition, see GCC Variable Argument Macros.

Restricted Pointers

The restrict keyword is a standard feature of the ISO/IEC 9899:1999 C standard, and is available as an extension in C89 and C++ modes.

The use of restrict is limited to the declaration of a pointer and specifies that the pointer provides exclusive initial access to the object to which it points. More simply, the restrict keyword is a way to identify that a pointer does not create an alias. Also, two different restricted pointers cannot designate the same object, and therefore, they are not aliases.

The compiler is free to use the information about restricted pointers and aliasing in order to better optimize C/C++ code that uses pointers. The restrict keyword is most useful when applied to function parameters about which the compiler would otherwise have little information about.

For example,

```
void fir(short *in, short *c, short *restrict out, int n);
```

The behavior of a program is undefined if it contains an assignment between two restricted pointers. Exceptions are:

* A function with a restricted pointer parameter may be called with an argument that is a restricted pointer.

* A function may return the value of a restricted pointer that is local to the function, and the return value may then be assigned to another restricted pointer.

If a program uses a restricted pointer in a way that it does not uniquely refer to storage, the behavior of the program is undefined.

Variable-Length Arrays

The compiler supports variable-length automatic arrays. This ISO/IEC 9899:1999 standard feature is also allowed as an extension in C89 mode (-c89). Variable-length arrays are not supported in C++ mode unless GNU C++ extensions are enabled using the -g++ switch.

Unlike other automatic arrays, variable-length arrays are declared with a non-constant length. This means that the space is allocated when the array is declared, and space is deallocated when the brace-level is exited.

The compiler does not allow jumping into the brace-level of the array and produces a compile-time error message if this is attempted. The compiler does allow breaking or jumping out of the brace-level, and it de-allocates the array when this occurs.

You can use variable-length arrays as function arguments, such as:

```
void var_array (int array_len, char data[array_len][array_len])
{
    /* code using data[][] */
}
```

The variable used for the array length must be in scope, and must have been previously declared.

The compiler calculates the length of an array at the time of allocation. It then remembers the array length until the brace-level is exited and can return it as the result of the `sizeof()` function performed on the array.

As an example, if you were to implement a routine for computation of a product of three matrices, you need to allocate a temporary matrix of the same size as input matrices. Declaring an automatic variable-size matrix is more convenient than allocating it from a heap. Note, however, that variable-length arrays are allocated on the stack, which means that sufficient stack space must be available.

The expression declares an array with a size that is computed at runtime. The length of the array is computed on entry to the block and saved in case `sizeof()` is applied to the array. For multi-dimensional arrays, the boundaries are also saved for address computation. After leaving the block, all the space allocated for the array and size information is deallocated.

For example, the following program prints 40, not 50:

```
#include <stdio.h>

void foo(int);

main ()
{
    foo(40);
}

void foo (int n)
{
    char c[n];
    n = 50;
```

```
    printf("%d", sizeof(c));
}
```

Non-Constant Initializer Support

The compiler does not require the elements of an aggregate initializer for an automatic variable to be constant expressions. This is a standard feature of the ISO/IEC 9899:1999 C standard and the ISO/IEC 14882:2003 C++ standard. The compiler supports it as an extension in C89 mode.

The following example shows an initializer with elements that vary at runtime.

```
void initializer (float a, float b)
{
    float the_array[2] = { a-b, a+b };
}
```

All automatic structures can be initialized by arbitrary expressions involving literals, previously declared variables, and functions.

Designated Initializers

This is a standard feature of the ISO/IEC 9899:1999 C standard. The compiler supports it as an extension in C89 and C++ modes, except for initializing arrays in C++11 mode.

This feature lets you specify the elements of an array or structure initializer in any order by specifying their designators-the array indices or structure field names to which they apply. All designators must be constant expressions, even in automatic arrays.

For an array initializer, the syntax [INDEX] appearing before an initializer element value specifies the index initialized by that value. Subsequent initializer elements are then applied to the sequentially following elements of the array, unless another use of the [INDEX] syntax appears. The index values must be constant expressions, even when the array being initialized is automatic.

The following example shows equivalent array initializers-the first in C89 form (without using the extension) and the second in C99 form, using the designators. Note that the [INDEX] designator precedes the value being assigned to that element.

```
/* Example 1 C Array Initializer */
/* C89 array initializer (no designators) */
int a[6] = { 0, 0, 15, 0, 29, 0 };

/* Equivalent C99 array initializer (with designators) */
int a[6] = { [4] 29, [2] 15 };
```

You can combine this technique of designated elements with initialization of successive non-designated elements. The two instructions below are equivalent. Note that any non-designated initial value is assigned to the next consecutive element of the structure or array.

```
/* Example 2 Mixed Array Initializer */
/* C89 array initializer (no designators) */
int a[6] = { 0, v1, v2, 0, v4, 0 };
```

```
/* Equivalent C99 array initializer (with designators) */
int a[6] = { [1] v1, v2, [4] v4 };
```

The following example shows how to label the array initializer elements when the designators are characters or
`enum` type.

```
/* Example 3 C Array Initializer With enum Type Indices */
/* C99 C array initializer (with designators) */
int whitespace[256] = {
    [' '] 1, ['\t'] 1, ['\v'] 1, ['\f'] 1, ['\n'] 1, ['\r'] 1
};


enum { e_ftp = 21, e_telnet = 23, e_smtp = 25, e_http = 80, e_nntp = 119 };
char *names[] = {
    [e_ftp] "ftp",
    [e_http] "http",
    [e_nntp] "nntp",
    [e_smtp] "smtp",
    [e_telnet] "telnet"
};
```

In a structure initializer, specify the name of the field to initialize with *fieldname*: before the element value. The
C89 and C99 struct initializers in the example below are equivalent.

```
/* Example 4 struct Initializer */
/* C89 struct Initializer (no designators) */
struct point {int x, y;};
struct point p = {xvalue, yvalue};


/* Equivalent C99 struct Initializer (with designators) */
struct point {int x, y;};
struct point p = {y: yvalue, x: xvalue};
```

Hexadecimal Floating-Point Numbers

This is a standard feature of the ISO/IEC:9899 1999 C standard . The compiler supports this as an extension in
C89 mode and in C++ mode.

Hexadecimal floating-point numbers have the following syntax.

```
hexadecimal-floating-constant:
   {0x|0X} hex-significand binary-exponent-part [floating-suffix]
hex-significand: hex-digits [ . [hex-digits]]
binary-exponent-part: {p|P} [+|-] decimal-digits
floating-suffix: { f | l | F | L }
```

The `hex-significand` is interpreted as a hexadecimal rational number. The digit sequence in the exponent
part is interpreted as a decimal integer. The exponent indicates the power of two by which the significand is to be
scaled. The floating suffix has the same meaning that it has for decimal floating constants: a constant with no suffix

is of type `double`, a constant with suffix F is of type `float`, and a constant with suffix L is of type `long double`.

Hexadecimal floating constants enable the programmer to specify the exact bit pattern required for a floating-point constant. For example, the declaration

```
float f = 0x1p-126f;
```

causes `f` to be initialized with the value `0x800000`.

Declarations Mixed With Code

In C89 mode, the compiler accepts declarations placed in the middle of code. This allows the declaration of local variables to be placed at the point where they are required. Therefore, the declaration can be combined with initialization of the variable. This is a standard feature of the ISO/IEC 9899:1999 C standard and the ISO/IEC 14882:2003 C++ standard.

For example, in the following function the declaration of `d` is delayed until its initial value is available, so that no variable is uninitialized at any point in the function.

```
void func(Key k) {
   Node *p = list;
   while (p && p->key != k)
      p = p->next;
   if (!p)
       return;
   Data *d = p->data;
   while (*d)
      process(*d++);
}
```

Compound Literals Support

This is a standard feature of the ISO/IEC:9899 1999 standard. The compiler supports it as an extension in C89 mode. Compound literals are not supported in C++ mode unless GNU C++ extensions are enabled using the -g++ switch.

The following example shows an ISO/IEC 9899:1990 standard C `struct` usage, followed by an equivalent ISO/IEC 9899:1999 standard C code that has been simplified using a compound literal.

```
/* Standard C89/C++ code*/
struct foo {int a; char b[2];};
struct foo make_foo(int x, char *s)
{
  struct foo temp;
  temp.a = x;
  temp.b[0] = s[0];
  if (s[0] !=  '\0')
     temp.b[1] = s[1];
  else
     temp.b[1] = '\0';
```

```
   return temp;
}

/* Standard C99 code*/
struct foo{ int a; char b[2];};
struct foo make_foo(int x, char *s)
{
   return((struct foo) {x, {s[0], s[0] ? s[1] : '\0'}});
}
```

C++ Style Comments

The compiler accepts C++ comments, beginning with `//` and ending at the end of the line, as in C programs. This comment representation is essentially compatible with standard C, except for the following case.

```
a = b
//* highly unusual */ c
;
```

which a standard C compiler processes as:

```
a = b/c;
```

and a C++ compiler and `ccblkfn` process as:

```
a = b;
```

Enumeration Constants That Are Not int Type

The CCES compiler allows enumeration constants to be integer types other than `int`, such as `unsigned int`, `long long`, or `unsigned long long`. See Enumeration Type Implementation Details for more information.

Boolean Type Support Keywords (bool, true, false)

The compiler supports a Boolean data type, `bool`, with values `true` and `false`. This is a standard feature of the ISO/IEC 14882:2003 C++ standard, and is available as a standard feature in the ISO/IEC 9899:1999 C standard when the `stdbool.h` header is included. It is supported as an extension in C89 mode, and as an extension in C99 mode when the `stdbool.h` header is not included.

The `bool` keyword is a unique signed integral type. There are two built-in constants of this type: `true` and `false`. When converting a numeric or pointer value to `bool`, a zero value becomes `false`, and a nonzero value becomes `true`. A `bool` value may be converted to `int` by promotion, taking `true` to one and `false` to zero. A numeric or pointer value is converted automatically to `bool` when needed.

Native Fixed-Point Types fract and accum

The compiler has support for the native fixed-point types `fract` and `accum` as defined by Chapter 4 of the "*Extensions to support embedded processors*" ISO/IEC draft technical report TR 18037. A discussion of how to use this support is given in Using Native Fixed-Point Types.

Inline Assembly Language Support Keyword (asm)

The compiler's `asm()` construct is used to code Blackfin assembly language instructions within a C/C++ function and to pass declarations and directives to the assembler. Use the `asm()` construct to express assembly language statements that cannot be expressed easily or efficiently with C/C++ constructs.

Using `asm()`, you can code complete assembly language instructions and specify the operands of the instruction using C expressions. When specifying operands with a C/C++ expression, you do not need to know which registers or memory locations contain C/C++ variables.

> **NOTE:** The compiler *does not analyze* code defined with the `asm()` construct— it passes this code directly to the assembler. The compiler performs substitutions for operands of the formats %0 through %9; however, it passes *everything else* to the assembler without reading or analyzing it. This means that the compiler cannot apply any enabled workarounds for silicon errata that may be triggered either by the contents of the `asm()` construct, or by the sequence of instructions formed by the `asm()` construct and the surrounding code produced by the compiler.

> **NOTE:** `asm()` constructs with inputs, outputs or affected registers are executable statements, and as such, may not appear before declarations within C/C++ functions in MISRA-C mode. `asm()` constructs may also be used at global scope, outside function declarations; such `asm()` constructs are used to pass declarations and directives directly to the assembler. They are not executable constructs, and may not have any inputs or outputs, or affect any registers.

> **NOTE:** When optimizing, the compiler sometimes changes the order in which generated functions appear in the output assembly file. However, if global-scope `asm()` constructs are placed between two function definitions, the compiler ensures that the function order is retained in the generated assembly file. Consequently, function inlining may be inhibited.

A simplified `asm()` construct without operands takes the following form.

```
asm(" NOP; ");
```

The complete assembly language instruction, enclosed in double quotes, is the argument to `asm()`. Using `asm()` constructs with operands requires additional syntax.

> **NOTE:** The compiler generates a label before and after inline assembly instructions when generating debug code. (See -g switch.) These labels are used to generate the debug line information used by the debugger. If the inline assembler inserts conditionally assembled code, an undefined symbol error is likely to occur at link-time. For example, the following code could cause undefined symbols if `MACRO` is undefined:
> ```
> asm("#ifdef MACRO");
> asm(" // assembly statements");
> asm("#endif");
> ```

If the inline assembler changes the current section and thereby causes the compiler labels to be placed in another section, such as a data section (instead of the default code section), then the debug line information will be incorrect for these lines.

The construct syntax is described in the following sections.

- asm() Construct Syntax

- Assembly Construct Operand Description

- Using long long Types in asm Constraints

- Assembly Constructs With Multiple Instructions

- Assembly Construct Reordering and Optimization

- Assembly Constructs With Input and Output Operands

- Assembly Constructs With Compile-Time Constants

- Assembly Constructs and Flow Control

- Guidelines for Using asm() Statements

asm() Construct Syntax

Use the following general syntax for asm() constructs.

```
asm [volatile] (
    template
    [:[constraint(output operand)[,constraint(output operand)]]
        [:[constraint(input operand)[,constraint(input operand)]]
            [:clobber string]]]
);
```

The syntax elements are defined as follows:

template

The template is a string containing the assembly instruction(s) with %number, indicating where the compiler should substitute the operands. Operands are numbered in order of occurrence from left to right, starting at 0. Separate multiple instructions with a semicolon; then enclose the entire string within double quotes. For more information on templates containing multiple instructions, see Assembly Constructs With Multiple Instructions.

constraint

The constraint is a string that directs the compiler to use certain groups of registers for the input and output operands. Enclose the constraint string within double quotes. For more information on operand constraints, see Assembly Construct Operand Description.

output operand

The output operands are the names of C/C++ variables that receive output from corresponding operands in the assembly instructions.

input operand

The input operand is a C/C++ expression that provides an input to a corresponding operand in the assembly instruction.

clobber string

The clobber string notifies the compiler that a list of registers is overwritten by the assembly instructions. Use lowercase characters to name clobbered registers. Enclose each name within double quotes, and separate each quoted register name with a comma. The input and output operands are guaranteed not to use any of the clobbered registers, so you can read and write the clobbered registers as often as you like.

See the **Register Names for asm() Constructs** table in Assembly Construct Operand Description for the list of individual registers. See the **Clobbered Register Sets** table in #pragma regs_clobbered *string* for the list of register sets.

It is vital that any register overwritten by an assembly instruction and not allocated by the constraints is listed in the clobber list. The list must include `memory` if an assembly instruction accesses memory.

asm() Construct Syntax Rules

These rules apply to assembly construct template syntax.

- The template is the only mandatory argument to `asm()`. All other arguments are optional.

- An operand constraint string followed by a C/C++ expression in parentheses describes each operand. For output operands, it must be possible to assign to the expression; that is, the expression must be legal on the left side of an assignment statement.

- A colon separates:

    - The template from the first output operand

    - The last output operand from the first input operand

    - The last input operand from the clobbered registers

- A space must be placed between adjacent colon field delimiters in order to avoid a clash with the C++ "::" reserved global resolution operator.

- A comma separates operands and registers within arguments.

- The number of operands in arguments must match the number of operands in your template.

- The maximum permissible number of operands is ten (%0, %1, %2, %3, %4, %5, %6, %7, %8, and %9).

   NOTE: The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. The compiler does not parse the assembler instruction template, does not interpret the template, and does not verify whether the template contains valid input for the assembler.

asm() Construct Template Example

The following example shows how to apply the `asm()` construct template to the Blackfin assembly language assignment instruction.

```
{
    int result, x;
```

```
    ...
    asm (
        "%0=%1;" :
        "=d" (result) :
        "d" (x)
    );
}
```

In the example above, note that:

- The template is `"%0=%1;"`. The `%0` is replaced with operand zero (`result`). The first operand, `%1`, is replaced with operand one (`x`).

- The output operand is the C/C++ variable `result`. The letter `d` is the operand constraint for the variable. This constrains the output to a data register, `R{0-7}`. The compiler generates code to copy the output from the data register to the variable `result`, if necessary. The `=` in `=d` indicates that the operand is an output.

- The input operand is the C/C++ variable `x`. The letter `d` in the operand constraint position for this variable constrains `x` to a data register, `R{0-7}`. If `x` is stored in a different kind of register or in memory, the compiler generates code to copy the value into a data register before the `asm()` construct uses it.

Assembly Construct Operand Description

The second and third arguments to the `asm()` construct describe the operands in the assembly language template. Several pieces of information must be conveyed for the compiler to know how to assign registers to operands. This information is conveyed with an operand constraint. The compiler needs to know what kind of registers the assembly instructions can operate on, so it can allocate the correct register type.

You convey this information with a letter in the operand constraint string that describes the class of allowable registers.

The *asm() Operand Constraints* table describes the correspondence between constraint letters and register classes.

> **NOTE:** The use of any letter not listed in the results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter.

To assign registers to the operands, the compiler must also be informed of which operands in an assembly language instruction are inputs, which are outputs, and which outputs may not overlap inputs. The compiler is told this in three ways.

- The output operand list appears as the first argument after the assembly language template. The list is separated from the assembly language template with a colon. The input operands are separated from the output operands with a colon and they always follow the output operands.

- The operand constraints describe which registers are modified by an assembly language instruction. The "=" in `=c` constraint indicates that the operand is an output; all output operand constraints must use =. Operands that are input-outputs must use "+". (See below.)

- The compiler may allocate an output operand in the same register as an unrelated input operand, unless the output or input operand has the `&=` constraint modifier. This situation can occur because the compiler assumes the inputs are consumed before the outputs are produced.

  This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use `&=` for each output operand that must not overlap an input or supply an `&` for the input operand.

Operand constraints indicate the kind of operand they describe by means of preceding symbols. Preceding symbols include: no symbol, =, +, &, ?, and #.

- *(no symbol)*

  The operand is an input. It must appear as part of the third argument to the `asm()` construct. The allocated register is loaded with the value of the C/C++ expression before the `asm()` template is executed. Its C/C++ expression is not modified by the `asm()` construct, and its value may be a constant or literal. Example: `d`

- *= symbol*

  The operand is an output. It must appear as part of the second argument to the `asm()` construct. Once the `asm()` template has been executed, the value in the allocated register is stored into the location indicated by its C/C++ expression; therefore, the expression must be one that would be valid as the left-hand side of an assignment. Example: `=d`

- *+ symbol*

  The operand is both an input and an output. It must appear as part of the second argument to the `asm()` construct. The allocated register is loaded with the C/C++ expression value, the `asm()` template is executed, and then the allocated register's new value is stored back into the C/C++ expression. Therefore, as with pure outputs, the C/C++ expression must be one that is valid on the left-hand side of an assignment. Example: `+d`

- *? symbol*

  The operand is temporary. It must appear as part of the third argument to the `asm()` construct. A register is allocated as working space for the duration of the `asm()` template execution. The register's initial value is undefined, and the register's final value is discarded. The corresponding C/C++ expression is not loaded into the register, but must be present. This expression is normally specified using a literal zero. Example: `?d`

- *& symbol*

  This operand constraint may be applied to inputs and outputs. It indicates that the register allocated to the input (or output) may not be one of the registers that are allocated to the outputs (or inputs). This operand constraint is used when one or more output registers are set while one or more inputs are yet to be referenced. (This situation sometimes occurs if the `asm()` template contains more than one instruction.) Example: `&d`

- *# symbol*

  The operand is an input, but the register's value is clobbered by the `asm()` template execution. The compiler may make no assumptions about the register's final value. An input operand with this constraint will not be

allocated the same register as any other input or output operand of the `asm()`. The operand must appear as part of the second argument to the `asm()` construct. Example: `#d`

The *asm() Operand Constraints* table lists the registers that may be allocated for each register constraint letter. The use of any letter not listed in the "Constraint" column of this table results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter. The *Register Names for asm() Constructs* table lists the registers that may be named as part of the clobber list.

It is also possible to claim registers directly, instead of requesting a register from a certain class using the constraint letters. You can claim the registers directly by simply naming the register in the location where the class letter would be. The register names are the same as those used to specify the clobber list; see the *Register Names for asm() Constructs* table.

The following example loads `sum` into `A0`, loads `x` and `y` into two `DREG` halves, executes the operation, and then stores the new total from `A0` back into `sum`.

```
asm("%0 += %1 * %2;"
    :"+a0"(sum)        /* output */
    :"H"(x),"H"(y)    /* input  */
);
```

NOTE: Naming registers in this way allows the `asm()` construct to specify several registers that must be related, such as the DAG registers for a circular buffer. This also allows the use of registers not covered by the register classes accepted by the `asm()` construct.

Table 2-25: asm() Operand Constraints

Constraint	Register Type	Registers
a	General addressing registers	P0 - P5
p	General addressing registers	P0 - P5
i	DAG addressing registers	I0 - I3
b	DAG addressing registers	I0 - I3
d	General data registers	R0 - R7
r	General data registers	R0 - R7
D	General data registers	R0 - R7
A	Accumulator registers	A0, A1
e	Accumulator registers	A0, A1
f	Modifier registers	M0 - M3
E	Even general data registers	R0, R2, R4, R6
O	Odd general data registers	R1, R3, R5, R7
h	High halves of the general data registers	R0.H, R1.H... R7.H
l	Low halves of the general data registers	R0.L, R1.L... R7.L

Table 2-25: asm() Operand Constraints (Continued)

Constraint	Register Type	Registers
H	Low or high halves of the general data registers	`R0.L, R1.L... R7.L`
L	Loop counter registers	`LC0, LC1`
I	General data register pairs	`(R0 - R1), (R2 - R3), (R4 - R5), (R6 - R7)`
n	None. For more information, see Assembly Constructs With Compile-Time Constants.	
constraint	Indicates the constraint is an input operand	
=constraint	Indicates the constraint is applied to an output operand	
&constraint	Indicates the constraint is applied to an input operand that may not be overlapped with an output operand	
=&constraint	Indicates the constraint is applied to an output operand that may not overlap an input operand	
?constraint	Indicates the constraint is temporary	
+constraint	Indicates the constraint is both an input and output operand	
#constraint	Indicates the constraint is an input operand whose value will be changed	

Table 2-26: Register Names for asm() Constructs

Clobber String	Meaning
`"r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7"`	General data register
`"p0", "p1", "p2", "p3", "p4", "p5"`	General addressing register
`"i0","i1", "i2", "i3"`	DAG addressing register
`"m0", "m1", "m2", "m3"`	Modify register
`"b0", "b1", "b2", "b3"`	Base register
`"l0", "l1", "l2", "l3"`	Length register
`"astat"`	ALU status register
`"seqstat"`	Sequencer status register
`"rets"`	Subroutine address register
`"cc"`	Condition code register
`"a0", "a1"`	Accumulator result register
`"lc0", "lc1"`	Loop counter register
`"r1:0", "r3:2", "r5:4", "r7:6"`	General data register pair
`"memory"`	Unspecified memory location(s)

Using long long Types in asm Constraints

It is possible to use an `asm()` constraint to specify a `long long` value, in which case the compiler will claim a valid register pair. The syntax for operands within the template is extended to allow the suffix "H" for the top 32 bits of the operand and the suffix "L" for the bottom 32 bits of the operand. A `long long` type is represented by the constraint letter "I".

For example,

```
long long int res;
{
    long long result64, x64 = 123;
    asm(
        "%0H = %1H; %0L = %1L;" :
        "=I" (result64) :
        "I" (x64)
    );
    res = result64;
}
```

In this example, the template is `"%0H=%1H; %0L=%1L;"`. The `%0H` is replaced with the register containing the top 32 bits of operand zero (`result64`), and `%0L` is replaced with the register containing the bottom 32 bits of operand zero (`result64`). Similarly, `%1H` and `%1L` are replaced with the registers containing the top 32 bits and bottom 32 bits, respectively, of operand one (`x64`).

Assembly Constructs With Multiple Instructions

There can be many assembly instructions in one template. Normal rules for line-breaking apply. In particular, the statement may spread over multiple lines. You are recommended not to split a string over more than one line, but to use the C language's string concatenation feature.

If you are placing the inline assembly statement in a preprocessor macro, see Compound Macros.

This is an example of multiple instructions in a template:

```
/* (pseudo code) r7 = x; r6 = y; result = x + y; */
    asm ("r7=%1;"
    "r6=%2;"
    "%0=r6+r7;"
    : "=d" (result)              /* output   */
    : "d" (x), "d" (y)           /* input    */
    : "r7", "r6");               /* clobbers */
```

> **NOTE:** Do not attempt to produce multiple-instruction asm constructs via a sequence of single-instruction asm constructs, as the compiler is not guaranteed to maintain the ordering.

For example, avoid the following:

```
/* BAD EXAMPLE: Do not use sequences of single-instruction
** asms. Use a single multiple-instruction asm instead.   */

asm("r7=%0;" : : "d" (x) : "r7");
```

```
asm("r6=%0;" : : "d" (y) : "r6");
asm("%0=r6+r7;" : "=d" (result));
```

Assembly Construct Reordering and Optimization

For the purpose of optimization, the compiler assumes that the side effects of an `asm()` construct are limited to changes in the output operands or the items specified using the clobber specifiers. This does not mean that you cannot use instructions with side effects, but be careful to notify the compiler that you are using them by using the clobber specifiers. (See the *Register Names for asm() Constructs* table in Assembly Construct Operand Description.)

The compiler may eliminate supplied assembly instructions (if the output operands are not used), move them out of loops, or reorder them with respect to other statements, where there is no visible data dependency. Also, if the instruction has a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register .

Use the keyword `volatile` to prevent an `asm()` instruction from being moved or deleted. For example,

```
#define set_priority(x) \
asm volatile ("STI %0;":    /* no outs */ : "d" (x))
```

A sequence of `asm volatile()` constructs is not guaranteed to be completely consecutive; it may be moved across jump instructions or in other ways that are not significant to the compiler. To force the compiler to keep the output consecutive, use one `asm volatile()` construct only, or use the output of the `asm()` construct in a C/C++ statement.

Assembly Constructs With Input and Output Operands

When an `asm` construct has both inputs and outputs, there are two aspects to consider:

1. Whether a value read from an input variable will be written back to the same variable or a different variable, on output.

2. Whether the input and output values will reside in the same register or different registers.

The most common case is when both input and output variables and input and output registers are different. In this case, the `asm` construct reads from one variable into a register, performs an operation which leaves the result in a different register, and writes that result from the register into a different output variable.

```
asm("%0 = %1;" : "=p" (newptr) : "p" (oldptr));
```

When the input and output variables are the same, the input and output registers are usually the same register. In this case, use the "+" constraint.

```
asm("%0 += 4;" : "+p" (sameptr));
```

When the input and output variables are different, but the input and output registers have to be the same (usually because of requirements of the assembly instructions), you indicate this to the compiler by using a different syntax for the input's constraint. Instead of specifying the register or class to be used, specify the output to which the input must be matched.

For example,

```
asm("%0 += 4;"
    :"=p" (newptr)        // an output, given a preg,
                          // stored into newptr.
    :"0" (oldptr));       // an input, given same reg as %0,
                          // initialized from oldptr
```

This specifies that the input `oldptr` has 0 (zero) as its constraint string, which means it must be assigned the same register as `%0` (`newptr`).

Assembly Constructs With Compile-Time Constants

The *n* input constraint informs the compiler that the corresponding input operand should not have its value loaded into a register. Instead, the compiler is to evaluate the operand, and then insert the operand's value into the assembly command as a literal numeric value. The operand must be a compile-time constant expression.

For example,

```
int r;
int arr[100];
asm("%0 = %1;" : "=d" (r) : "d" (sizeof(arr)));   // "d" constraint
```

produces code similar to:

```
R0 = 400 (X);    // compiler loads value into register
R1 = R0;         // compiler replaces %1 with register
```

whereas:

```
int r;
int arr[100];
asm("%0 = %1;" : "=d" (r) : "n" (sizeof(arr)));   // "n" constraint
```

produces code similar to:

```
R1 = 400;         // compiler replaces %1 with value
```

If the expression is not a compile-time constant, the compiler gives an error:

```
int r; int arr[100];
asm("%0 = %1;" : "=d" (r) : "n" (arr));
                      // error: operand
                      // for "n" constraint
                      // must be a compile-time constant
```

Assembly Constructs and Flow Control

> **NOTE:** Do not place flow-control operations within an `asm()` construct that "leaves" the `asm()` construct, such as calling a procedure or performing a jump to another piece of code that is not within the `asm()` construct itself.

For example, the compiler is careful to adhere to the calling conventions for preserved registers when making a procedure call. If an `asm()` construct calls a procedure, the `asm()` construct must also ensure that all conventions are obeyed, or the called procedure may corrupt the state used by the function containing the `asm()` construct.

It is also inadvisable to use labels in `asm()` statements, especially when function inlining is enabled. If a function containing such `asm` statements is inlined more than once in a file, there will be multiple definitions of the label, resulting in an assembler error. If possible, use PC-relative jumps in `asm` statements.

Guidelines for Using asm() Statements

Certain operations are performed more efficiently using other compiler features, and result in source code that is more clear and easier to read.

Accessing System Registers

System registers are accessed most efficiently using the functions in `sysreg.h` instead of using `asm()` statements (see also System Built-In Functions).

Accessing Memory-Mapped Registers (MMRs)

MMRs can be accessed using the macros in the `cdef*.h` files (for example, `cdefBF531.h`) that are supplied with CCES (see also Memory-Mapped Register Access Built-In Functions).

Memory Banks

By default, the compiler assumes that all memory may be accessed with equal performance, but this is not always the case: some parts of your application may be in faster internal memory, and other parts in slower, external memory. The compiler supports the concept of *memory banks* to group code or data with equivalent performance characteristics. By providing this information to the compiler, you can improve the performance of your application.

Memory Banks Versus Sections

Note that memory banks are different from sections:

- Section is a "hard" directive, using a name that is meaningful to the linker. If the `.ldf` file does not map the named section, a linker error occurs.

- A memory bank is a "soft" informational characterization, using a name that is not visible to the linker. The compiler uses optimization to take advantage of the bank's performance characteristics. However, if the `.ldf` file maps the code or data to memory that performs differently, the application still functions (albeit with a possible reduction in performance).

Pragmas and Qualifiers

Memory banks may be referenced through both memory bank pragmas and memory bank qualifiers:

- Use memory bank pragmas to specify the memory banks used by all the code or data of a function. For example,

```
#pragma data_bank(bank_external)
int *getptr(void) { return ptr2; }
```

- Use memory bank qualifiers to specify the memory bank referenced by individual variables. For example,

```
int bank("bank_internal") *ptr1;
int bank("bank_external") *ptr2;
```

Memory Bank Selection

The compiler applies the following process for determine which bank is being referenced.

Memory Banks for Code

The compiler uses the following process for deducing the memory bank which contains instructions:

1. If the function is immediately preceded by `#pragma code_bank( bank )`, the function's instructions are considered to reside in memory bank `bank`.

2. If the function is immediately preceded by `#pragma code_bank` or `#pragma code_bank()`, the function's instructions are not considered to reside in any defined memory bank.

3. Otherwise, if `#pragma default_code_bank( defbank )` has been used in the compilation unit prior to the definition of the function, the function's instructions are considered to reside in memory bank `defbank`.

4. Otherwise, the function's instructions are not considered to reside in any defined memory bank.

For more information, see #pragma code_bank(*bankname*).

Memory Banks for Data

The compiler uses the following process for deducing which memory bank contains variables that are auto storage class:

1. If the variable declaration includes a memory bank qualifier, for example,

   ```
   int bank("bank") x;
   ```

   then the variable will be considered to reside in bank `bank`.

2. Otherwise, if the function is immediately preceded by `#pragma stack_bank( bank )`, then the variable is considered to reside in memory bank `bank`.

3. Otherwise, if the function is immediately preceded by `#pragma stack_bank` or `#pragma stack_bank()`, then the variable is not considered to reside in any memory bank.

4. Otherwise, if `#pragma default_stack_bank( defbank )` has been used in the compilation unit prior to the definition of the function, the variable is considered to reside in memory bank `defbank`.

5. Otherwise, the variable is not considered to reside in any defined memory bank.

For more information, see #pragma stack_bank(*bankname*).

The compiler uses the following process for selecting the memory bank to contain `static` variables defined within a function:

1. If the variable declaration includes a memory bank qualifier, for example,

```
static int bank( "bank" ) x;
```

then the variable will be considered to reside in bank *bank*.

2. Otherwise, if the function is immediately preceded by `#pragma data_bank( bank )`, then the variable is considered to reside in memory bank *bank*.

3. Otherwise, if the function is immediately preceded by `#pragma data_bank` or `#pragma data_bank()`, then the variable is not considered to reside in any memory bank.

4. Otherwise, if `#pragma default_data_bank( defbank )` has been used in the compilation unit prior to the definition of the function, the variable is considered to reside in memory bank *defbank*.

5. Otherwise, the variable is not considered to reside in any defined memory bank.

For more information, see #pragma data_bank(*bankname*).

The compiler uses the following process for selecting the memory bank to contain variables defined at global scope:

1. If the variable declaration includes a memory bank qualifier, for example,

```
static int bank( "bank" ) x;
```

then the variable will be considered to reside in bank *bank*.

2. Otherwise, if `#pragma default_data_bank( defbank )` has been used in the compilation unit prior to the definition of the variable, the variable is considered to reside in memory bank *defbank*.

3. Otherwise, the variable is not considered to reside in any defined memory bank.

The identified memory bank is used for pointer dereferences. For example,

```
#pragma data_bank(bank_external)
int f(int *a, int *b) {
   return *a + *b;  // *a and *b both considered to be
}                   // loads from "bank_external"
```

For more information, see #pragma default_data_bank(bankname).

Performance Characteristics

You can specify the performance characteristics of a memory bank. This will allow the compiler to generate optimal code when accessing the bank. You can specify the following characteristics:

- Cycles required to read the memory bank. Use #pragma bank_read_cycles(*bankname, cycles[, bits]*) to specify this characteristic.

- Cycles required to write the memory bank. Use #pragma bank_write_cycles(*bankname, cycles*[, bits]) to specify this characteristic.

- The maximum bit width supported by accesses to the memory bank. Use #pragma bank_maximum_width(*bankname, width*) to specify this characteristic.

Memory Bank Kinds

Each memory bank has a defined kind. The memory bank kinds supported on Blackfin processors are listed in the *Memory Bank Kinds* table. Not all kinds are available on all processors.

Table 2-27: Memory Bank Kinds

Memory Bank Kind	Description
internal	Corresponds to L1 Instr SRAM
L2	Corresponds to on-processor, off-core memory.
L2_cached	Corresponds to on-processor, off-core memory that is cached in L1.
external	Corresponds to memory that is external to the processor.
external_cached	Corresponds to memory that is external to the processor, but cached in L1.

Predefined Banks

The compiler predefines a memory bank for each supported memory bank kind, using the same name but with a `bank_` prefix. For example, the following uses the internal and external memory banks:

```
#pragma code_bank("bank_external")
int next_counter(void) {
    static int bank("bank_internal") counter;
    return counter++; }
```

These predefined memory banks have predefined performance characteristics, such as read and write cycle counts, that are appropriate for the memory kind. You can override these performance characteristics via pragmas. For more information, see Memory Bank Pragmas.

The memory bank kinds are listed in the *Memory Bank Kinds* table.

Defining Additional Banks

New memory banks are defined when first used, whether this happens in a memory bank pragma, or in a memory bank qualifier. When created, memory banks have kind `internal`, unless otherwise specified by `#pragma memory_bank_kind`.

The compiler does not attach any significance to the name of any new memory banks you create.

Placement Support Keyword (section)

The `section()` keyword directs the compiler to place an object or function in an assembly `.SECTION` of the compiler's intermediate output file. You name the assembly `.SECTION` with the string literal parameter of the `section()` keyword. If you do not specify a `section()` keyword for an object or function declaration, the compiler uses a default section. The `.ldf` file supplied to the linker must also be updated to support the additional named section. For information on the default sections, see Memory Section Usage.

Applying `section()` is meaningful only when the data item is something that the compiler can place in the named section. Apply `section()` only to top-level, named objects that have static duration (for example, objects that are explicitly `static`, or are given as external-object definitions).

The following example shows the definition of a static variable that is placed in the section called `bingo`.

```
static section("bingo") int x;
```

The `section()` keyword has the limitation that section initialization qualifiers cannot be used within the section name string. The compiler may generate labels containing this string, which will result in assembly syntax errors. Additionally, the keyword is not compatible with any pragmas that precede the object or function. For finer control over section placement and compatibility with other pragmas, use `#pragma section`.

Refer to #pragma section/#pragma default_section for more information.

> **NOTE:** The `section` keyword replaces the `segment` keyword in earlier releases of the compiler. Although the `segment()` keyword is supported by the compiler of the current release, Analog Devices recommends that you revise legacy code.

Placement of Compiler-Generated Code and Data

If the `section()` keyword is not used, the compiler emits code and data into default sections. The `-section` switch (-section *id=section_name[, id=section_name...]*) can be used to specify alternatives for these defaults on the command-line, and the #pragma section/#pragma default_section can be used to specify alternatives for some of them within the source file.

In addition, when using certain features of C/C++, the compiler may be required to produce internal data structures. The `-section` switch and the `default_section` pragma allow you to override the default location where the data would be placed.

For example, the following command instructs the compiler to place all the C++ virtual function look-up tables into the `vtbl_data` section, rather than the default `vtbl` section.

```
ccblkfn -section vtbl=vtbl_data test.cpp -c++
```

> **NOTE:** It is the user's responsibility to ensure that appropriately named sections exist in the `.ldf` file.

When both `-section` switches and `default_section` pragmas are used, the default_section pragmas take priority.

Long Identifiers

The compiler supports C identifiers of up to 1022 characters in length; C++ identifiers typically have a slightly shorter limit, as the limit applies to the identifier after *name mangling* is used to transform it into a suitable symbol for linking, and for C++, some of the symbol space is required to represent the identifier's type.

Compiler Built-In Functions

The compiler supports built-in functions (sometimes called *intrinsics*) that enable efficient use of hardware resources. These functions are:

- builtins.h

- Fractional Value Built-In Functions

- ETSI Support

- fract16 and fract32 Literal Values

- Converting Between Fractional and Floating-Point Values

- Complex Fractional Built-In Functions in C

- Changing the RND_MOD Bit

- Complex Operations in C++

- Packed 16-Bit Integer Built-In Functions

- Division Functions

- Full-Precision Accumulator Built-In Functions

- Viterbi History and Decoding Functions

- Search Built-in Functions

- Circular Buffer Built-In Functions

- Endian-Swapping Intrinsics

- System Built-In Functions

- Cache Built-In Functions

- Compiler Performance Built-In Functions

- Video Operation Built-In Functions

- Misaligned Data Built-In Functions

- Memory-Mapped Register Access Built-In Functions

Knowledge of these functions is built into the `ccblkfn` compiler. Your program uses them via normal function call syntax. The compiler notices the invocation and generates one or more machine instructions, just as it does for normal operators, such as + and *.

Built-in functions have names that begin with `__builtin_`. Note that identifiers beginning with double underscores (`__`) are reserved by the C standard, so these names will not conflict with user program identifiers.

These functions are specific to individual architectures, and the following sections list built-in functions currently supported on Blackfin processors. Various system header files provide definitions and access to the intrinsics as described below.

builtins.h

The `builtins.h` header file defines prototypes for all built-in functions supported by the compiler; include this header file in any module that invokes a built-in function.

The header file also defines short names for each built-in function: for each built-in function `__builtin_func()`, the header file defines the short name `func()`. These short names can be disabled selectively or as a group, by defining macros prior to include the header file. The *Macros Controlling builtins.h* table lists these macros.

Table 2-28: Macros Controlling builtins.h

Macro Name	Effect
`__NO_SHORTNAMES`	If defined, prevents any short names from being defined.
`__SPECIFIC_NAMES`	If defined, short name *func* is defined only if corresponding macro `__ENABLE_FUNC` is defined.
`__ENABLE_FUNC`	Causes short name *func* to be defined, if `__SPECIFIC_NAMES` is also defined.
`__DISABLE_FUNC`	Prevents short name *func* from being defined.
`__DEFINED_FUNC`	Multiple-inclusion guard. The header file defines this macro when it defines short name *func*, but does not define short name *func* if this macro is defined already.

Fractional Value Built-In Functions

Two approaches may be used to access the fractional arithmetic and the parallel 16-bit operations supported by the Blackfin processor instructions. One is to use the native fixed-point types `fract` and `accum`. This approach is discussed in Using Native Fixed-Point Types. Alternatively, built-in functions may be used to specify fractional operations. This section discusses the use of these built-in functions.

Table 2-29: Fractional Value Data Types

C Type	Usage
`fract16`	Single 16-bit signed fractional value, typedef to short
`fract32`	Single 32-bit signed fractional value, typedef to long
`fract64`	Single 64-bit signed fractional value, typedef to long long
`fract`	Single 16-bit signed fractional value, native type
`long fract`	Single 32-bit signed fractional value, native type
`fract2x16`	Double 16-bit signed fractional value
`dpf16`	Part of a 32-bit `dpf32` value.

Table 2-29: Fractional Value Data Types (Continued)

C Type	Usage
dpf32	A 32-bit value composed from two dpf16 values, *hi* and *lo*: dpf32 = (hi<<16) + (lo<<1)

The various data types used in the built-in functions described in this section are defined; see the ***Types Used in Accumulator Built-In Functions*** table, Accumulator Built-In Function Prototypes.

NOTE: See Data Storage Formats for information on how fract16, fract32, fract, long fract, and fract2x16 types are represented. See the *Blackfin Processor Programming Reference* manual for information on saturation, rounding (biased and unbiased), and truncating.

Because fractional arithmetic uses slightly different instructions to normal arithmetic, you cannot normally use the standard C operators on the fract16 and fract32 data types and get the right result. Instead, use the built-in functions described here to work with fractional data.

The fract.h header file provides access to the definitions for each of the built-in functions that support fractional values. These functions have names with suffixes _fr1x16 for single fract16, _fr2x16 for dual fract16, and _fr1x32 for single fract32. All the functions in fract.h are marked as inline, so when compiling with the compiler optimizer, the built-in functions are inlined.

NOTE: The 16-bit fractional shift built-in functions without _clip in the name ignore all but the least significant five bits of the shift magnitude. The 32-bit fractional shift built-in functions without _clip in the name ignore all but the least significant 6 bits of the shift magnitude. The _clip variants of these built-in functions automatically clip the shift magnitude to within a 5- or 6-bit range. For example, where a 5-bit (-16..+15) range is required, the _clip variants would clip the value +63 to be +15, while the non-_clip variant would discard the upper bits and interpret bit 5 as the sign bit, giving a value of -1. To avoid unexpected results, use the _clip variants of the functions unless the shift magnitude is known to be within the 5- or 6-bit range.

See 16-Bit Fractional Built-In Functions for descriptions of built-in functions that work primarily with fract16 data. See 32-Bit Fractional Built-In Functions for descriptions of built-in functions that work primarily with fract32 data.

See fract2x16 Built-In Functions for descriptions of built-in functions that work primarily with fract2x16 data. Note that when compiling programs that use the single data fract16 operations, the compiler optimizer attempts to automatically detect cases where parallel operations can be performed. In other words, re-coding an algorithm to make explicit use of fract2x16 built-in functions in place of the fract1x16 ones does not always yield a performance benefit.

See ETSI Support for information on mapping the European Telecommunications Standards Institute (ETSI) fract functions onto the compiler built-in functions.

16-Bit Fractional Built-In Functions

All the built-in functions described here are saturating unless otherwise stated. These built-ins operate primarily on the `fract16` and `fract` types although one of the multiplies returns a `fract32`.

The following built-in functions are available.

```
fract16 add_fr1x16(fract16 f1, fract16 f2);
fract add_fx1x16(fract f1, fract f2);
```

Performs 16-bit addition of the two input parameters (`f1+f2`). The `fract` version is included for completeness only; it is exactly equivalent to the + operator on `fract` types.

```
fract16 sub_fr1x16(fract16 f1, fract16 f2);
fract sub_fx1x16(fract f1, fract f2);
```

Performs 16-bit subtraction of the two input parameters (`f1-f2`). The `fract` version is included for completeness only; it is exactly equivalent to the − operator on `fract` types.

```
fract16 mult_fr1x16(fract16 f1, fract16 f2);
fract mult_fx1x16(fract f1, fract f2);
```

Performs 16-bit multiplication of the input parameters (`f1*f2`). The result is truncated to 16 bits. The `fract` version is exactly equivalent to the * operator on `fract` types in the truncation rounding mode.

```
fract16 multr_fr1x16(fract16 f1, fract16 f2);
fract multr_fx1x16(fract f1, fract f2);
```

Performs a 16-bit fractional multiplication (`f1*f2`) of the two input parameters. The result is rounded to 16 bits. Whether the rounding is biased or unbiased depends on what the `RND_MOD` bit in the `ASTAT` register is set to. The `fract` version is exactly equivalent to the * operator on `fract` types when the biased or unbiased rounding mode is used.

```
fract32 mult_fr1x32(fract16 f1, fract16 f2);
long fract mult_fx1x32(fract f1, fract f2);
```

Performs a fractional multiplication on two 16-bit fractions, returning the 32-bit result. The `fract` version is included for completeness only; it is exactly equivalent to writing `(long fract)f1 * (long fract)f2`.

```
fract16 abs_fr1x16(fract16 f1);
fract abs_fx1x16(fract f1);
```

Returns the 16-bit value that is the absolute value of the input parameter. Where the input is `0x8000`, saturation occurs and `0x7fff` is returned. The `fract` version is included for completeness only; it is exactly equivalent to the `absr` function.

```
fract16 min_fr1x16(fract16 f1, fract16 f2);
fract min_fx1x16(fract f1, fract f2);
```

Returns the minimum of the two input parameters.

```
fract16 max_fr1x16(fract16 f1, fract16 f2);
fract max_fx1x16(fract f1, fract f2);
```

Returns the maximum of the two input parameters.

```
fract16 negate_fr1x16(fract16 f1);
fract negate_fx1x16(fract f1);
```

Returns the 16-bit result of the negation of the input parameter (-f1). If the input is 0x8000, saturation occurs and 0x7fff is returned. The fract version is included for completeness only; it is exactly equivalent to writing -f1.

```
fract16 shl_fr1x16(fract16 src, short shft);
fract shl_fx1x16(fract src, short shft);
```

Arithmetically shifts the src variable left by shft places. The empty bits are zero-filled. If shft is negative, the shift is to the right by abs(shft) places with sign extension.

```
fract16 shl_fr1x16_clip(fract16 src, short shft);
fract shl_fx1x16_clip(fract src, short shft);
```

Arithmetically shifts the src variable left by shft (clipped to 5 bits) places. The empty bits are zero filled. If shft is negative, the shift is to the right by abs(shft) places with sign extension.

```
fract16 shr_fr1x16(fract16 src, short shft);
fract shr_fx1x16(fract src, short shft);
```

Arithmetically shifts the src variable right by shft places with sign extension. If shft is negative, the shift is to the left by abs(shft) places, and the empty bits are zero-filled.

```
fract16 shr_fr1x16_clip(fract16 src, short shft);
fract shr_fx1x16_clip(fract src, short shft);
```

Arithmetically shifts the src variable right by shft (clipped to 5 bits) places with sign extension. If shft is negative, the shift is to the left by abs(shft) places, and the empty bits are zero-filled.

```
fract16 shrl_fr1x16(fract16 src, short shft);
fract shrl_fx1x16(fract src, short shft);
```

Logically shifts the src variable right by shft places. There is no sign extension and no saturation-the empty bits are zero-filled.

```
fract16 shrl_fr1x16_clip(fract16 src, short shft);
fract shrl_fx1x16_clip(fract src, short shft);
```

Logically shifts the src variable right by shft (clipped to 5 bits) places. There is no sign extension and no saturation-the empty bits are zero-filled.

```
short norm_fr1x16(fract16 f1);
short norm_fx1x16(fract f1);
```

Returns the number of left shifts required to normalize the input variable so that it is either in the interval 0x4000 to 0x7fff, or in the interval 0x8000 to 0xc000. In other words,

```
fract16 x;
shl_fr1x16(x, norm_fr1x16(x));
```

Returns a value in the range 0x4000 to 0x7fff, or in the range 0x8000 to 0xc000, except in the special case where x is zero. The fract version is equivalent to the countlsr function.

32-Bit Fractional Built-In Functions

All the built-in functions described here are saturating unless otherwise stated. These built-in functions operate primarily on the `fract32` and `long fract` types, although there are a couple of functions that convert between 16- and 32-bit fractional types.

```
fract32 add_fr1x32(fract32 f1,fract32 f2);
long fract add_fx1x32(long fract f1,long fract f2);
```

Performs 32-bit addition of the two input parameters (`f1+f2`). The `long fract` version is included for completeness only; it is exactly equivalent to the + operator on `long fract` types.

```
fract32 sub_fr1x32(fract32 f1,fract32 f2);
long fract sub_fx1x32(long fract f1,long fract f2);
```

Performs 32-bit subtraction of the two input parameters (`f1-f2`). The `long fract` version is included for completeness only; it is exactly equivalent to the − operator on `long fract` types.

```
fract32 mult_fr1x32x32(fract32 f1,fract32 f2);
long fract mult_fx1x32x32(long fract f1,long fract f2);
```

Performs 32-bit multiplication of the input parameters (`f1*f2`). The result (which is calculated internally with an accuracy of 40 bits) is rounded (biased rounding) to 32 bits. You might also consider using the * operator on the `long fract` type in biased rounding mode. This provides better rounding precision and may offer comparable performance.

```
fract32 multr_fr1x32x32(fract32 f1,fract32 f2);
long fract multr_fx1x32x32(long fract f1,long fract f2);
```

Same as `mult_fr1x32x32` and `mult_fx1x32x32` but with additional rounding precision. You might also consider using the * operator on the `long fract` type in biased rounding mode, which offers comparable performance. The results may differ in the rounding performed.

```
fract32 mult_fr1x32x32NS(fract32 f1, fract32 f2);
long fract mult_fx1x32x32NS(long fract f1, long fract f2);
```

Performs 32-bit non-saturating multiplication of the input parameters (`f1*f2`). This is somewhat faster than `mult_fr1x32x32` or `mult_fx1x32x32`. The result (which is calculated internally with an accuracy of 40 bits) is rounded (biased rounding) to 32 bits. You might also consider using the * operator on the `long fract` type in biased rounding mode. This performs a saturating multiplication and gives a more precisely-rounded result at some cost of efficiency.

```
fract32 abs_fr1x32(fract32 f1);
long fract abs_fx1x32(long fract f1);
```

Returns the 32-bit value that is the absolute value of the input parameter. Where the input is `0x80000000`, saturation occurs and `0x7fffffff` is returned. The `long fract` version is included for completeness only; it is exactly equivalent to the `abslr` function.

```
fract32 min_fr1x32(fract32 f1, fract32 f2);
long fract min_fx1x32(long fract f1, long fract f2);
```

Returns the minimum of the two input parameters

```
fract32 max_fr1x32(fract32 f1, fract32 f2);
long fract max_fx1x32(long fract f1, long fract f2);
```

Returns the maximum of the two input parameters

```
fract32 negate_fr1x32(fract32 f1);
long fract negate_fx1x32(long fract f1);
```

Returns the 32-bit result of the negation of the input parameter (-f1). If the input is 0x80000000, saturation occurs and 0x7fffffff is returned. The long fract version is included for completeness only; it is exactly equivalent to writing -f1.

```
fract32 shl_fr1x32(fract32 src, short shft);
long fract shl_fx1x32(long fract src, short shft);
```

Arithmetically shifts the src variable left by shft places. The empty bits are zero filled. If shft is negative, the shift is to the right by abs(shft) places with sign extension.

```
fract32 shl_fr1x32_clip(fract32 src, short shft);
long fract shl_fx1x32_clip(long fract src, short shft);
```

Arithmetically shifts the src variable left by shft (clipped to 6 bits) places. The empty bits are zero filled. If shft is negative, the shift is to the right by abs(shft) places with sign extension.

```
fract32 shr_fr1x32(fract32 src, short shft);
long fract shr_fx1x32(long fract src, short shft);
```

Arithmetically shifts the src variable right by shft places with sign extension. If shft is negative, the shift is to the left by abs(shft) places, and the empty bits are zero-filled.

```
fract32 shr_fr1x32_clip(fract32 src, short shft);
long fract shr_fx1x32_clip(long fract src, short shft);
```

Arithmetically shifts the src variable right by shft (clipped to 6 bits) places with sign extension. If shft is negative, the shift is to the left by abs(shft) places, and the empty bits are zero-filled.

```
fract16 sat_fr1x32(fract32 f1);
fract sat_fx1x32(long fract f1);
```

If f1>0x00007fff, it returns 0x7fff. If f1<0xffff8000, it returns 0x8000. Otherwise, it returns the lower 16 bits of f1.

```
fract16 round_fr1x32(fract32 f1);
fract round_fx1x32(long fract f1);
```

Rounds the 32-bit fract to a 16-bit fract using biased rounding. The long fract version is equivalent to casting a long fract to fract in biased rounding mode.

```
int norm_fr1x32(fract32 f1);
int norm_fx1x32(long fract f1);
```

Returns the number of left shifts required to normalize the input variable so that it is either in the interval 0x40000000 to 0x7fffffff, or in the interval 0x80000000 to 0xc0000000. In other words,

```
fract32 x;
shl_fr1x32(x,norm_fr1x32(x));
```

Returns a value in the range `0x40000000` to `0x7fffffff`, or in the range `0x80000000` to `0xc0000000`, except in the special case where `x` is zero. The `long fract` version is equivalent to the `countlslr` function.

```
fract16 trunc_fr1x32(fract32 f1);
fract trunc_fx1x32(long fract f1);
```

Returns the top 16 bits of `f1`-it truncates `f1` to 16 bits. The `long fract` version is equivalent to casting a `long fract` to `fract` in truncation rounding mode.

fract2x16 Built-In Functions

All built-in functions described here are saturating unless otherwise stated. These built-ins operate primarily on the `fract2x16` type, although there are composition and decomposition functions for the `fract2x16` type, multi-plies that return `fract32` and `long fract` results, and operations on a single `fract2x16` pair that return `fract16` and `fract` types.

The notation used to represent two `fract16` or `fract` values packed into a `fract2x16` is `{a,b}`, where `a` is the `fract16` or `fract` packed into the high half, and `b` is the `fract16` or `fract` packed into the low half. A `fract2x16` can be thought of as two `fract16`s or two `fract`s as the representation of the two types is the same.

```
fract2x16 compose_fr2x16(fract16 f1, fract16 f2);
fract2x16 compose_fx_fr2x16(fract f1, fract f2);
```

Takes two 16-bit fractional values and returns a `fract2x16` value.

Input: two `fract16` or `fract` values

Returns: `{f1,f2}`

```
fract16 high_of_fr2x16(fract2x16 f);
fract high_of_fx_fr2x16(fract2x16 f);
```

Takes a `fract2x16` and returns the "high half" `fract16` or `fract`.

Input: `f{a,b}`

Returns: a

```
fract16 low_of_fr2x16(fract2x16 f);
fract low_of_fx_fr2x16(fract2x16 f);
```

Takes a `fract2x16` and returns the "low half" `fract16` or `fract`

Input: `f{a,b}`

Returns: b

```
fract2x16 add_fr2x16(fract2x16 f1,fract2x16 f2);
```

Adds two packed fracts.

Input: `f1{a,b} f2{c,d}`

Returns: `{a+c,b+d}`

```
fract2x16 sub_fr2x16(fract2x16 f1,fract2x16 f2);
```

Subtracts two packed fracts.

Input: `f1{a,b} f2{c,d}`

Returns: `{a-c,b-d}`

```
fract2x16 mult_fr2x16(fract2x16 f1,fract2x16 f2);
```

Multiplies two packed fracts. Truncates the results to 16 bits.

Input: `f1{a,b} f2{c,d}`

Returns: `{trunc16(a*c),trunc16(b*d)}`

```
fract2x16 multr_fr2x16(fract2x16 f1,fract2x16 f2);
```

Multiplies two packed fracts. Rounds the result to 16 bits. Whether the rounding is biased or unbiased depends on what the `RND_MOD` bit in the `ASTAT` register is set to.

Input: `f1{a,b} f2{c,d}`

Returns: `{round16{a*c},round16{b*d}}`

```
fract2x16 negate_fr2x16(fract2x16 f1);
```

Negates both 16-bit fracts in the packed `fract`. If one of the `fract16` values is `0x8000`, saturation occurs and `0x7fff` is the result of the negation.

Input: `f1{a,b}`

Returns: `{-a,-b}`

```
fract2x16 shl_fr2x16(fract2x16 f1,short shft);
```

Arithmetically shifts both `fract16s` in the `fract2x16` left by `shft` places, and returns the packed result. The empty bits are zero-filled. If `shft` is negative, the shift is to the right by `abs(shft)` places with sign extension.

Input: `f1{a,b} shft`

Returns: `{ashft,bshft}`

```
fract2x16 shl_fr2x16_clip(fract2x16 f1,short shft);
```

Arithmetically shifts both `fract16s` in the `fract2x16` left by `shft` (clipped to 5 bits) places and returns the packed result. The empty bits are zero filled. If `shft` is negative, the shift is to the right by `abs(shft)` places with sign extension.

```
fract2x16 shr_fr2x16(fract2x16 f1, short shft);
```

Arithmetically shifts both `fract16`s in the `fract2x16` right by `shft` places with sign extension and returns the packed result. If `shft` is negative, the shift is to the left by `abs(shft)` places and the empty bits are zero-filled.

Input: `f1{a,b} shft`

Returns: `{a>>shft,b>>shft}`
```
fract2x16 shr_fr2x16_clip(fract2x16 f1, short shft);
```

Arithmetically shifts both `fract16`s in the `fract2x16` right by `shft` (clipped to 5 bits) places with sign extension, and returns the packed result. If `shft` is negative, the shift is to the left by `abs(shft)` places and the empty bits are zero-filled.
```
fract2x16 shrl_fr2x16(fract2x16 f1, short shft);
```

Logically shifts both `fract16`s in the `fract2x16` right by `shft` places. There is no sign extension and no saturation-the empty bits are zero-filled.

Input: `f1{a,b} shft`

Returns: `{a>>shft,b>>shft}` //logical shift
```
fract2x16 shrl_fr2x16_clip(fract2x16 f1, short shft);
```

Logically shifts both `fract16`s in the `fract2x16` right by `shft` places (clipped to 5 bits). There is no sign extension and no saturation-the empty bits are zero-filled.
```
fract2x16 abs_fr2x16(fract2x16 f1);
```

Returns the absolute value of both `fract16`s in the `fract2x16`.

Input: `f1{a,b}`

Returns: `{abs(a),abs(b)}`
```
fract2x16 min_fr2x16(fract2x16 f1, fract2x16 f2);
```

Returns the minimums of the two pairs of `fract16`s in the two input `fract2x16s`.

Input: `f1{a,b} f2{c,d}`

Returns: `{min(a,c),min(b,d)}`
```
fract2x16 max_fr2x16(fract2x16 f1, fract2x16 f2);
```

Returns the maximums of the two pairs of `fract16`s in the two input `fract2x16s`.

Input: `f1{a,b} f2{c,d}`

Returns: `{max(a,c),max(b,d)}`
```
fract16 sum_fr2x16(fract2x16 f1) fract sum_fx_fr2x16(fract2x16 f1)
```

Performs a sideways addition of the two `fract16`s or `fract`s in f1.

Input: `f1{a,b}`

Returns: `a+b`

```
fract2x16 add_as_fr2x16(fract2x16 f1, fract2x16 f2);
```

Performs a vector add/subtract on the two input `fract2x16s`.

Input: `f1{a,b} f2{c,d}`

Returns: `{a+c,b-d}`

```
fract2x16 add_sa_fr2x16(fract2x16 f1, fract2x16 f2);
```

Performs a vector subtract/add on the two input `fract2x16s`.

Input: `f1{a,b} f2{c,d}`

Returns: `{a-c,b+d}`

```
fract16 diff_hl_fr2x16(fract2x16f1);
fract diff_hl_fx_fr2x16(fract2x16f1);
```

Takes the difference (high-low) of the two `fract16s` or `fracts` in the `fract2x16`.

Input: `f1{a,b}`

Returns: `a-b`

```
fract16 diff_lh_fr2x16(fract2x16 f1);
fract diff_lh_fx_fr2x16(fract2x16 f1);
```

Takes the difference (low-high) of the two `fract16s` or `fracts` in the `fract2x16`.

Input: `f1{a,b}`

Returns: `b-a`

```
fract32 mult_ll_fr2x16(fract2x16 f1, fract2x16 f2);
long fract mult_ll_fx_fr2x16(fract2x16 f1, fract2x16 f2);
```

Cross-over multiplication. Multiplies the low half of `f1` with the low half of `f2`.

Input: `f1{a,b} f2{c,d}`

Returns: `(fract32) b*d` or `(long fract) b*d`

```
fract32 mult_hl_fr2x16(fract2x16 f1, fract2x16 f2);
long fract mult_hl_fx_fr2x16(fract2x16 f1, fract2x16 f2);
```

Cross-over multiplication. Multiplies the high half of `f1` with the low half of `f2`.

Input: `f1{a,b} f2{c,d}`

Returns: `(fract32) a*d` or `(long fract) a*d`

```
fract32 mult_lh_fr2x16(fract2x16 f1, fract2x16 f2);
long fract mult_lh_fx_fr2x16(fract2x16 f1, fract2x16 f2);
```

Cross-over multiplication. Multiplies the low half of `f1` with the high half of `f2`.

Input: `f1{a,b} f2{c,d}`

Returns: `(fract32) b*c` or `(long fract) b*c`

```
fract32 mult_hh_fr2x16(fract2x16 f1, fract2x16 f2);
long fract mult_hh_fx_fr2x16(fract2x16 f1, fract2x16 f2);
```

Cross-over multiplication. Multiplies the high half of `f1` with the high half of `f2`.

Input: `f1{a,b} f2{c,d}`

Returns: `(fract32) a*c` or `(long fract) a*c`

ETSI Support

In addition to the native fixed-point types as defined by the ISO/IEC draft technical report Technical Report 18037 that the compiler supports (Using Native Fixed-Point Types), CCES also provides support for a set of functions that manipulate fixed-point data using operations defined by the European Telecommunications Standards Institute (ETSI). These operations (or "macros") are a set of functions for performing fixed-point, bit-accurate, arithmetic-they were initially defined by ETSI in 1993 for the standardization of the half-rate GSM speech code and are also used to define the GSM enhanced full-rate (EFR) and adaptive multi-rate (AMR) speech codecs. The ETSI functions supported by CCES are defined in the header file `libetsi.h`.

One of the features of the ETSI functions is the support they include to detect overflow and carry. When overflow or a carry occurs, an ETSI function sets one of the appropriate global variables `Overflow` or `Carry`. The global variables are defined in the ETSI library and are declared in the header file `libetsi.h` as:

```
extern int Overflow;
extern int Carry;
```

The `Overflow` and `Carry` flags are *sticky*, which means that the ETSI functions will only set the global variables but will not unset them. It is your responsibility to ensure the variables are reset between operations.

The default behavior of the ETSI functions is, however, to disable detection of overflow and carry. This means that more efficient versions of the functions can be provided, often as inline code that the compiler can expand and insert directly into the code stream that it generates, thus avoiding the overheads associated with a function call.

In general, the definition of the ETSI functions assume intermediate results will be rounded using biased rounding. Some inline versions of the ETSI functions will therefore be affected by the `RND_MOD` flag in the ASTAT register. For bit-exact results, set the `RND_MOD` flag to provide biased rounding. For more information, see Changing the RND_MOD Bit.

If an application wants to enable carry and overflow detection in the ETSI functions, then it must ensure that the macro `__SET_ETSI_FLAGS` is defined before it includes the libetsi.h header file-one way of doing this is by using the compiler switch `-D__SET_ETSI_FLAGS`.

CCES provides two different versions of the ETSI library:

- `libetsi.dlb` - this library has been built with support for carry and overflow detection disabled for optimal performance. This is the default ETSI library that will be used when linking an application.

- `libetsico.dlb` - this version of the library has full support for carry and overflow detection. To link against this library, specify the compiler switch `-letsico`. Note that the ETSI functions in this library are not thread-safe.

If the macro `RENAME_ETSI_NEGATE` is defined, the ETSI function `negate` is renamed to `etsi_negate`. This is useful because the C++ Standard declares a template function called `negate` (found in the C++ include `functional`).

By default, the following ETSI shift functions conform to the ETSI definition by clipping the second parameter to the size of the first parameter:

```
fract16 shl (fract16 var1, short var2);
fract16 shr (fract16 var1, short var2);
fract32 L_shl (fract32 L_var1, short var2);
fract32 L_shr (fract32 L_var1, short var2);
```

If macro `__SET_ETSI_FLAGS` is not set to 1 (see above), then faster versions of these shift functions are available that do not clip the second parameter and only use the least significant five bits of the second parameter as the shift count. The faster versions will be used if the macro `_ADI_FAST_ETSI` is defined, either before including `libetsi.h` in the source of the application, or by using the compiler's `-D` switch.

The following routines are available in the ETSI library. These routines are commonly classified into three groups:

- Those that return or primarily operate on 32-bit fractional values in double-precision format (DPF).

- Those that return or primarily operate on 32-bit fractional values in 1.31 format.

- Those that return or primarily operate on 16-bit fractional values in 1.15 format.

32-Bit Fractional ETSI Routines Using Double-Precision Format

Double-precision format (DPF) is represented as:

```
dpf32 = (hi<<16) + (lo<<1)
```

where:

- `dpf32` is a 32-bit signed integer (`typedef`'d to fract32, itself a `typedef` to `int`)

- `hi` and `lo` are 16-bit signed integers (`typedef`'d to `short`)

- `hi` contains the 16 most-significant bits of a 32-bit fractional value, and `lo` contains the next 15 bits as a signed value.

A 32-bit DPF value ranges from 0x80000000 to 0x7ffffffe.

The following two functions, which are defined in `libetsi.h`, convert a `dpf32`-type value into a `fract32`, and a `fract32` into a `dpf32`-type value respectively:

```
fract32 dpf32_to_fract32(dfp32 x);
dpf32 fract32_to_dpf32(fract32 x);
```

(A call to these functions compiles into a simple assignment statement.)

The ETSI operations that use DPF are:

```
dpf32 L_Comp(dpf16 hi, dpf16 lo);
```

Composes a 32-bit value from the given high and low DPF components. The sign is provided with the low half, and the result is calculated as:

```
(hi>>16) + (lo>>1);
void L_Extract(dpf32 src, dpf16 *hi, dpft16 *lo);
```

Extracts low and high halves of a 32-bit value into 16-bit DPF component values pointed to by the hi and lo parameters. The values calculated are:

```
*hi = bit16 to bit31 of src
*lo = (src - (hi<<16))>>1

dpf32 Mpy_32(dpf16 hi1, dpf16 lo1, dpf16 hi2, dpf16 lo2);
```

Performs the multiplication of two 32-bit values, each provided as high and low DPF components. The result returned is calculated as:

```
Res = L_mult(hi1, hi2);
Res = L_mac(Res, mult(hi1, lo2), 1);
Res = L_mac(Res, mult(lo1, hi2), 1);
dpf32 Mpy_32_16(dpf16 hi, dpf16 lo, fract16 v);
```

Multiplies the parameter v, which is a `fract16` value, by a 32-bit DPF value provided as high and low halves, and returns the result as a 32-bit value.

```
dpf32 Div_32(dpf32 L_num, dpf16 denom_hi, dpf16 denom_lo);
```

Performs a 32-bit fractional division using a 32-bit dividend (`L_num`) and a 32-bit DPF divisor (`denom_hi` and `denom_lo`). Both the dividend and the divisor must be positive fractional values. Also, the value of the dividend must be less than the value of the divisor, and the value of the divisor must not be less than `0x40000000` (which is equivalent to the value 0.5).The result of `Div_32` is accurate to 24 bits of precision.

Use of these functions typically requires fractional data to be converted to and from DPF. The `L_Extract()` and `L_Comp()` functions can be used for this purpose.

An example that uses these DPF operators follows. The example implements a 32-bit fractional multiplication (also implemented by the compiler built-in function `mult_fr1x32x32()`).

```
#include <libetsi.h>

fract32 mul32by32_etsi(fract32 a, fract32 b)
{
    dpf32 exp_prec_res;
    dpf16 a_hi, a_lo;
    dpf16 b_hi, b_lo;
    fract32 res;
    /* Extract two 16-bit DPF components from a 32-bit fract */
    L_Extract(a, &a_hi, &a_lo);
    L_Extract(b, &b_hi, &b_lo);
    /* 32-bit extended precision multiply */
```

```
    exp_prec_res = Mpy_32(a_hi, a_lo, b_hi, b_lo);
    /* Convert from a dpf32 to fract32 - a simple assignment */
    res = dpf32_to_fract32(exp_prec_res);
    /* return result */
    return res;
}
```

32-Bit Fractional ETSI Routines Using 1.31 Format

The following functions return or primarily operate on 32-bit fractional data, in 1.31 format.

```
fract32 L_add_c(fract32 a, fract32 b);
```

Performs a 32-bit addition of the two input parameters. Saturation occurs if the sum overflows or underflows. If the sum overflows, the function returns 0x7fffffff; if it underflows, the function returns 0x80000000. When linking with the library libetsico.dlb, saturation will cause the Overflow flag to be set, while the Carry flag will be set if a carry was detected.

```
fract32 L_abs(fract32 a);
```

Returns the 32-bit absolute value of the input parameter. In cases where the input is equal to `0x80000000`, saturation occurs and `0x7fffffff` is returned.

```
fract32 L_add(fract32 a, fract32 b);
```

Returns the 32-bit saturated result of the addition of the two input parameters. If overflow occurs, the Overflow flag will be set provided that the application is linked with the library libetsico.dlb.

```
fract32 L_deposit_h(fract16 hi);
```

Deposits the 16-bit parameter into the 16 most significant bits of the 32-bit result. The least-significant 16 bits are set to zero.

```
fract32 L_deposit_l(fract16 lo);
```

Deposits the 16-bit parameter into the 16 least significant bits of the 32-bit result. The most significant bits are sign-extended for the input.

```
fract32 L_mac(fract32 acc, fract16 f1, fract16 f2);
```

Performs a fractional multiplication of the two 16-bit parameters and returns the saturated sum of the product and the 32-bit parameter.

```
fract32 L_macNs(fract32 Lf,fract16 f1, fract16 f2);
```

Performs a non-saturating version of the L_mac operation. When linking with the library libetsico.dlb, the Overflow and Carry flags are set if a carry or overflow/underflow occurs.

```
fract32 L_mls (fract32 Lf, fract16 f);
```

Multiplies both the most significant bits and the least significant bits of the 32-bit parameter Lf, by the 16-bit parameter f.

```
fract32 L_msu(fract32 Lf, fract16 f1, fract16 f2);
```

Performs a fractional multiplication of the two 16-bit parameters and returns the saturated difference between the product and the 32-bit parameter.

```
fract32 L_msuNs(fract32 Lf, fract16 f1, fract16 f2);
```

Performs a non-saturating version of the `L_msu` operation. When linking with the library `libetsico.dlb`, the `Overflow` and `Carry` flags are set if a carry or overflow/underflow occurs.

```
fract32 L_mult(fract16 f1, fract16 f2);
```

Returns the 32-bit saturated result of the fractional multiplication of the two 16-bit parameters.

```
fract32 L_negate(fract32 Lf);
```

Returns the 32-bit result of the negation of the parameter. Where the input parameter is `0x80000000` saturation occurs and `0x7fffffff` is returned.

```
fract32 L_sat(fract32 Lf);
```

Returns `0x80000000` if `Carry` and `Overflow` flags are set (corresponding to an underflow condition); otherwise, if `Overflow` is set, returns `0x7fffffff`. The default version of the function simply returns `Lf` as no checking or setting of the `Overflow` and `Carry` flags is performed.

```
fract32 L_shl(fract32 src, short shft);
```

Arithmetically shifts the 32-bit first parameter to the left by the value given in the 16-bit second parameter. The empty bits of the 32-bit value are zero-filled. If the shifting value, `shft`, is negative, the source is shifted to the right by `-shft` with sign-extension. The result is saturated in cases of overflow and underflow.

When linking with the library `libetsico.dlb`, the `Overflow` flag is set when overflow occurs.

> **NOTE:** To avoid unexpected results when the shift count exceeds the number of bits in the first parameter, the function normally clips the second parameter to the size of the first. If clipping is not required, then a faster version of the function is available, but only if the macro `__SET_ETSI_FLAGS` is not defined as 1. To use the faster version of the shift function that will use the least significant five bits of the second parameter as the shift count, define the macro `_ADI_FAST_ETSI` before including `libetsi.h`, or define it on the compile command-line.

```
fract32 L_shr(fract32 src, short shft);
```

Arithmetically shifts the 32-bit first parameter to the right by the value given in the 16-bit second parameter with sign extension. If the shifting value is negative, the source is shifted to the left. The result is saturated in cases of overflow and underflow.

When linking with the library `libetsico.dlb`, the `Overflow` flag is set when overflow occurs.

> **NOTE:** To avoid unexpected results when the shift count exceeds the number of bits in the first parameter, the function normally clips the second parameter to the size of the first. If clipping is not required, then a faster version of the function is available, but only if the macro `__SET_ETSI_FLAGS` is not defined as 1. To use the faster version of the shift function that will use the least significant five bits of the second

parameter as the shift count, define the macro `_ADI_FAST_ETSI` before including `libetsi.h`, or define it on the compile command-line.

```
fract32 L_shr_r(fract32 src, short shft);
```

Performs the shift-right operation as per `L_shr` but with rounding. When linking with the library `libetsico.dlb`, the `Overflow` and `Carry` flags are set if a carry or overflow/underflow occurs.

```
fract32 L_shift_r(fract32 src, short shft);
```

Arithmetically shifts the first parameter; if the second parameter is positive, then shift left. Otherwise, shift right. The result is rounded and then saturated if necessary.

```
fract32 L_sub(fract32 Lf1, fract32 Lf2);
```

Returns the 32-bit saturated result of the subtraction of two 32-bit parameters (`Lf1 - Lf2`).

```
fract32 L_sub_c(fract32 Lf1, fract32 Lf2);
```

Performs 32-bit subtraction of two fractional values (`Lf1 - Lf2`). When linking with the library `libetsico.dlb`, the `Carry` and `Overflow` flags are set if a carry and overflow/underflow occurs during subtraction.

16-Bit Fractional ETSI Routines

The following functions return or primarily operate on 16-bit fractional data.

```
fract16 abs_s(fract16 f);
```

Returns the 16-bit value that is the absolute value of the input parameter. When the input is `0x8000`, saturation occurs and `0x7fff` is returned.

```
fract16 add(fract16 f1, fract16 f2);
```

Returns the 16-bit sum of the two `fract16` input parameters.

Saturation occurs if the sum overflows or underflows. If the sum overflows, the function returns `0x7fff`; if it underflows, the function returns `0x8000`. When linking with the library `libetsico.dlb`, the `Overflow` and `Carry` flags are set when carry or overflow/underflow occurs.

```
fract16 div_l (fract32 L_num, fract16 den);
```

This function produces a result which is the fractional integer division of the first parameter by the second. Both inputs must be positive and the least significant word of the second parameter must be greater or equal to the first; the result is positive (leading bit equal to 0) and truncated to 16 bits. The function calls `abort()` on division error conditions.

```
fract16 div_s(fract16 f1, fract16 f2);
```

Returns the 16-bit result of the fractional integer division of `f1` by `f2`. Both `f1` and `f2` must be positive fractional values with `f2` greater than `f1`.

```
fract16 extract_l(fract32 Lf);
```

Returns the 16 least significant bits of the 32-bit parameter provided.

```
fract16 extract_h(fract32 Lf);
```

Returns the 16 most significant bits of the 32-bit parameter provided.

```
fract16 mac_r(fract32 acc, fract16 f1, fract16 f2);
```

Performs an `L_mac` operation using the three parameters provided. The result is the rounded 16 most significant bits of the 32-bit results from the `L_mac` operation.

```
fract16 msu_r(fract32 Lf, fract16 f1, fract16 f2);
```

Performs an `L_msu` operation using the three parameters provided. The result is the rounded 16 most significant bits of the 32-bit result from the `L_msu` operation.

```
fract16 mult(fract16 f1, fract16 f2);
```

Returns the 16-bit result of the fractional multiplication of the input parameters. The result is saturated.

```
fract16 mult_r(fract16 f1, fract16 f2);
```

Performs a 16-bit multiply with rounding of the result of the fractional multiplication of the two input parameters.

> **NOTE:** The compiler generates the following 16-bit fractional multiply instruction for this function:
>
> ```
> Rx.L = Ry.L * Rz.L;
> ```
>
> This instruction's result is affected by the `RND_MOD` bit in the ASTAT register, which means that the results may not always be ETSI-compliant. To avoid this issue, set `RND_MOD` before using this function, or link against the library `libetsico.dlb`, which contains a version of the function that ensures the `RND_MOD` bit is set as necessary for the duration of the function's execution. For more information, see Changing the RND_MOD Bit.

```
fract16 negate(fract16 f);
```

Returns the 16-bit result of the negation of the input parameter. If the input is `0x8000`, saturation occurs and `0x7fff` is returned.

> **NOTE:** This function is renamed `etsi_negate` if the macro `RENAME_ETSI_NEGATE` is defined; this avoids a conflict with the C++ template function `negate` that is defined in the include file `functional`.

```
int norm_l(fract32 Lf);
```

Returns the number of left shifts required to normalize the input variable so that it is either in the interval `0x40000000` to `0x7fffffff`, or in the interval `0x80000000` to `0xc0000000`. In other words,

```
fract32 Lx;
fract32 Lxn = L_shl(Lx, norm_l(x));
```

sets `Lxn` to a value in the range `0x40000000` to `0x7fffffff`, or in the range `0x80000000` to `0xc0000000`, except in the special case where `Lx` is zero.

> **NOTE:** This function uses the Blackfin `SIGNBITS` instruction.

```
int norm_s(fract16 f);
```

Returns the number of left shifts required to normalize the input variable so that it is either in the interval `0x4000` to `0x7fff`, or in the interval `0x8000` to `0xc000`. In other words,

```
fract16 x; fract16 xn = shl(x, norm_s(x));
```

sets `xn` to a value in the range `0x4000` to `0x7fff`, or in the range `0x8000` to `0xc000`, except in the special case where `x` is zero.

> **NOTE:** This function uses the Blackfin `SIGNBITS` instruction.

```
fract16 round(fract32 Lf);
```

Rounds the lower 16 bits of the 32-bit input parameter into the most significant 16 bits with saturation. The resulting bits are shifted right by 16.

```
fract16 saturate(fract32);
```

Uses biased rounding with saturation to return the most significant 16 bits of the input parameter. If the input parameter is less than `0x8000`, `0x8000` is returned.

```
fract16 shl(fract16 src, short shft);
```

Arithmetically shifts the `src` variable left by `shft` places. The empty bits are zero-filled. If `shft` is negative, the shift is to the right by `shft` places.

When linking with the library `libetsico.dlb`, the `Overflow` and `Carry` flags are set when carry or overflow/underflow occurs.

> **NOTE:** To avoid unexpected results when the shift count exceeds the number of bits in the first parameter, the function normally clips the second parameter to the size of the first. If clipping is not required, then a faster version of the function is available, but only if the macro `__SET_ETSI_FLAGS` is not defined as 1. To use the faster version of the shift function that will use the least significant five bits of the second parameter as the shift count, define the macro `_ADI_FAST_ETSI` before including `libetsi.h`, or define it on the compile command-line.

```
fract16 shr(fract16 src, short shft);
```

Arithmetically shifts the `src` variable right by `shft` places with sign extension. If `shft` is negative, the shift is to the left by `shft` places.

When linking with the library `libetsico.dlb`, the `Overflow` and `Carry` flags are set when carry or overflow/underflow occurs. A built-in version of this function is also provided.

> **NOTE:** To avoid unexpected results when the shift count exceeds the number of bits in the first parameter, the function normally clips the second parameter to the size of the first. If clipping is not required, then a faster version of the function is available, but only if the macro `__SET_ETSI_FLAGS` is not defined as 1. To use the faster version of the shift function that will use the least significant five bits of the second parameter as the shift count, define the macro `_ADI_FAST_ETSI` before including `libetsi.h`, or define it on the compile command-line.

```
fract16 shr_r(fract16 src, short shft);
```

Performs a shift to the right as per the `shr()` operation with additional rounding and saturation of the result.

```
fract16 shift_r(fract16 src, short shft);
```

Arithmetically shifts the first parameter; if the second parameter is positive then shift left otherwise shift right; the result is rounded and then saturated if necessary.

```
fract16 sub(fract16 f1, fract16 f2);
```

Returns the 16-bit result of the subtraction of the two parameters (`f1 - f2`). Saturation occurs if the result overflows or underflows. If the result overflows, the function returns `0x7fff`; if it underflows, the function returns `0x8000`.

When linking with the library `libetsico.dlb`, the `Overflow` and `Carry` flags are set when carry or overflow/underflow occurs.

```
short i_mult(short v1, short v2);
```

Multiplies two 16-bit integers, with no saturation.

fract16 and fract32 Literal Values

This section discusses natural ways to define `fract16` and `fract32` literal values. For discussion of literals of the native fixed-point types `fract` and `accum`, see Native Fixed-Point Constants.

A constant with an `r` suffix is defined to be a native fixed-point constant of `fract` type. This should not be used to initialize a `fract16` or `fract32` constant since the type conversion yields an unexpected result (see Data Type Conversions and Fixed-Point Types for more details).

The suffixes `r32` and `r16` can be used in C mode to represent `fract32` and `fract16` literals. They allow users to naturally express literals in fractional format. These literals are represented as 32-bit signed integral types.

For example,

`0x4000` is the same as 0.5r16

`0x40000000` is the same as 0.5r32

These literals cannot be used in the expressions of the preprocessing directives `#if` or `#elif`.

> **NOTE:** Despite appearances, literal values expressed in this syntax are still "normal" integer values, and are subject to the usual rules of integer arithmetic and type promotion/conversion. Be sure to use the built-in functions if you require fractional arithmetic.

Converting Between Fractional and Floating-Point Values

The CCES run-time libraries contain high-level support for converting between fractional and floating-point values. The include file `fract2float_conv.h` defines functions which perform conversions between `fract16`, `fract32`, and `float` types.

The following functions are defined:

```
// Converting between fract16 and fract32
fract32 fr16_to_fr32(fract16);
```

```
fract16 fr32_to_fr16(fract32);

// Converting from float to fract16/fract32
fract32 float_to_fr32(float);
fract16 float_to_fr16(float);

// Converting from long double to fract16/fract32
fract32 long_double_to_fr32(long double);
fract16 long_double_to_fr16(long double);

// Converting from fract16/fract32 to float
float fr16_to_float(fract16);
float fr32_to_float(fract32);

// Converting from fract16/fract32 to long double
long double fr16_to_long_double(fract16);
long double fr32_to_long_double(fract32);
```

In addition, the following functions are defined for use on the native fixed-point types `fract` and `long fract`. These are provided for completeness only, as casts between the different types provide the same functionality.

```
// Convert between fract and long fract
long fract fx16_to_fx32(fract);
fract fx32_to_fx16(long fract);

// Convert from float to fract/long fract
long fract float_to_fx32(float);
fract float_to_fx16(float);

// Convert from long double to fract/long fract
long fract long_double_to_fx32(long double);
fract long_double_to_fx16(long double);

// Convert from fract/long fract to float
float fx16_to_float(fract);
float fx32_to_float(long fract);

// Convert from fract/long fract to long double
long double fx16_to_long_double(fract);
long double fx32_to_long_double(long fract);
```

The float-to-fract conversions are saturating such that the result lies in the range of the fractional data type.

These functions can be employed to aid implementation of critical parts of applications using fractional arithmetic that would otherwise use floating-point arithmetic. Such implementations usually requires data to be scaled into the fractional range before converting to `fract16` or `fract32`, and this is still true when using the functions defined in `fract2float_conv.h`.

Floating-Point Multiplication Using fracts

This example implements a floating-point multiplication using an ETSI `fract` implementation.

```
#include <fract2float_conv.h>
#include <fract_typedef.h>
#include <libetsi.h>
#include <math.h>

/* return a*b calculated using fract implementation */
float mul_fp(float a, float b) {
    float scaled_a, scaled_b, fract_div_res, result;
    int exp_a, exp_b, exp_res;
    fract32 fract_a, fract_b, fract_res;
    fract32 fract_exp_a, fract_exp_b, fract_exp_res;
    dpf16   hia, loa, hib, lob;

    /* if either input is 0, return 0 */
    if (a == 0.0 || b == 0.0)
        return 0.0;
     /* scale inputs */
        scaled_a = frexpf(a, &exp_a);
        scaled_b = frexpf(b, &exp_b);

     /* convert scaled inputs to fract */
        fract_a = float_to_fr32(scaled_a);
        fract_b = float_to_fr32(scaled_b);

    /* extract the 16-bit DPF words from the fract inputs */
        L_Extract(fract_a, &hia, &loa);
        L_Extract(fract_b, &hib, &lob);

    /* do fractional multiplication in extended precision */
        fract_res = Mpy_32(hia, loa, hib, lob);

    /* multiply exponents by adding */
        exp_res = exp_a + exp_b;

    /* convert mul result back to float */
        fract_div_res = fr32_to_float(fract_res);

    /* compose the floating-point result */
        result = ldexpf(fract_div_res, exp_res);

    /* return result */
        return result;
}  /* mul_fp */
```

Complex Fractional Built-In Functions in C

The `complex_int16` and `complex_int32` types are used to hold complex integer numbers. The `complex_int16` type contains real and imaginary values, both as 16-bit integral numbers:

```
typedef struct {
    short re, im;
} complex_int16;
```

The `complex_int32` type contains real and imaginary values, both as 32-bit integral numbers:

```
typedef struct {
    int re, im;
} complex_int32;
```

The `complex_fract16` and `complex_fract32` types are used to hold complex fractional numbers. The `complex_fract16` type contains real and imaginary values, both as 16-bit fractional numbers:

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

The `complex_fract32` type contains real and imaginary values, both as 32-bit fractional numbers:

```
typedef struct {
    fract32 re, im;
} complex_fract32;
```

The `complex_int16`, `complex_int32`, `complex_fract16` and `complex_fract32` types are defined by the `complex.h` header file. The `complex.h` header file also declares numerous library functions for manipulating complex fractional numbers, in addition to the built-in functions listed in this section. These functions are documented in the DSP Run-Time Library Reference chapter.

The compiler also supports the following built-in operations for complex fractional numbers. For each of these built-ins, fractional results values are rounded and saturated as required. The rounding mode is determined by the `RND_MOD` bit in the `ASTAT` register.

- The following built-in function generates instructions to calculate and return the complex fractional square of `a`.
  ```
  complex_fract16 csqu_fr16(complex_fract16 a);
  ```

- The following built-in functions generate instructions to calculate and return the complex fractional square magnitude of `a`.
  ```
  fract16 csqumag_fr16(complex_fract16 a);
  fract32 csqumagw_fr16(complex_fract16 a);
  acc40 A_csqumag_fr16(complex_fract16 a);
  ```

- The following functions can be used to extract the real and imaginary parts of the `complex_fract16` or `complex_fract32` input `a`. Note that the `re` and `im` fields of all the complex types defined above can also be accessed directly.
  ```
  fract16 real_fr16(complex_fract16 a);
  fract16 imag_fr16(complex_fract16 a);
  fract real_fx_fr16(complex_fract16 a);
  fract imag_fx_fr16(complex_fract16 a);
  fract32 real_fr32(complex_fract32 a);
  ```

```
fract32 imag_fr32(complex_fract32 a);
long fract real_fx_fr32(complex_fract32 a);
long fract imag_fx_fr32(complex_fract32 a);
```

- The following functions can be used to create a `complex_fract16` or `complex_fract32` type instance from two fractional inputs which correspond to the required result's real and imaginary parts. Again note that this can also be achieved by writing to the `re` and `im` fields of the type directly.

```
complex_fract16 ccompose_fr16 (fract16 real, fract16 imag);
complex_fract16 ccompose_fx_fr16 (fract real, fract imag);
complex_fract32 ccompose_fr32 (fract32 real, fract32 imag);
complex_fract32 ccompose_fx_fr32 (long fract real, long fract imag);
```

- The following functions perform a complex addition of the inputs and return the result.

```
complex_fract16 cadd_fr16(complex_fract16 a, complex_fract16 b);
complex_fract32 cadd_fr32(complex_fract32 a, complex_fract32 b);
```

- The following functions perform a complex subtraction of the inputs and return the result.

```
complex_fract16 csub_fr16(complex_fract16 a, complex_fract16 b);
complex_fract32 csub_fr32(complex_fract32 a, complex_fract32 b);
```

- The following functions perform a complex multiplication of the inputs.

```
complex_fract16 cmlt_fr16(complex_fract16 a, complex_fract16 b);
complex_fract32 cmlt_fr32(complex_fract32 a, complex_fract32 b);
```

- The following function returns the complex conjugate of the input.

```
complex_fract32 conj_fr32(complex_fract32 a);
```

A number of built-in functions are available to provide convenient access to the native hardware support for complex multiplies on the Blackfin+ series of processors. These built-in functions are not available for use on earlier Blackfin processors. The following built-ins are provided:

- The following built-in function generates instructions to calculate and return the complex fractional product of the inputs a and b. The low 16 bits of the resulting real and imaginary parts are rounded off and the results saturated to 16 bits to produce a `complex_fract16` result.

```
complex_fract16 cmul_fr16(complex_fract16 a, complex_fract16 b);
```

- The following built-in function generates instructions to calculate and return the complex fractional product of the input a and the complex conjugate of input b. The low 16 bits of the resulting real and imaginary parts are rounded off and the results saturated to 16 bits to produce a `complex_fract16` result.

```
complex_fract16 conj_cmul_fr16(complex_fract16 a, complex_fract16 b);
```

- The following built-in function generates instructions to calculate and return the complex fractional product of the complex conjugates of inputs a and b. The low 16 bits of the resulting real and imaginary parts are rounded off and the results saturated to 16 bits to produce a `complex_fract16` result.

```
complex_fract16 conj_conj_cmul_fr16(complex_fract16 a, complex_fract16 b);
```

- The following built-in function generates instructions to calculate and return the complex fractional product of the inputs a and b. The low 16 bits of the resulting real and imaginary parts are truncated and the results saturated to 16 bits to produce a `complex_fract16` result.

```
complex_fract16 cmult_fr16(complex_fract16 a, complex_fract16 b);
```

- The following built-in function generates instructions to calculate and return the complex fractional product of the input a and the complex conjugate of input b. The low 16 bits of the resulting real and imaginary parts are truncated and the results saturated to 16 bits to produce a `complex_fract16` result.

```
complex_fract16 conj_cmult_fr16(complex_fract16 a, complex_fract16 b);
```

- The following built-in function generates instructions to calculate and return the complex fractional product of the complex conjugates of inputs a and b. The low 16 bits of the resulting real and imaginary parts are truncated and the results saturated to 16 bits to produce a `complex_fract16` result.

```
complex_fract16 conj_conj_cmult_fr16(complex_fract16 a, complex_fract16 b);
```

- The following built-in function generates instructions to calculate and return the complex fractional product of the inputs a and b. The real and imaginary parts of the product are saturated to 32 bits to produce a `complex_fract32` result.

```
complex_fract32 cmulw_fr16(complex_fract16 a, complex_fract16 b);
```

- The following built-in function generates instructions to calculate and return the complex fractional product of the input a and the complex conjugate of input b. The real and imaginary parts of the product are saturated to 32 bits to produce a `complex_fract32` result.

```
complex_fract32 conj_cmulw_fr16(complex_fract16 a, complex_fract16 b);
```

- The following built-in function generates instructions to calculate and return the complex fractional product of the complex conjugates of inputs a and b. The real and imaginary parts of the product are saturated to 32 bits to produce a `complex_fract32` result.

```
complex_fract32 conj_conj_cmulw_fr16(complex_fract16 a, complex_fract16 b);
```

- The following built-in function generates instructions to calculate and return the complex integral product of the inputs a and b. The real and imaginary parts of the product are saturated to 16 bits to produce a `complex_int16` result.

```
complex_int16 cmul_i16(complex_int16 a, complex_int16 b);
```

- The following built-in function generates instructions to calculate and return the complex integral product of the input a and the complex conjugate of input b. The real and imaginary parts of the product are saturated to 16 bits to produce a `complex_int16` result.

```
complex_int16 conj_cmul_i16(complex_int16 a, complex_int16 b);
```

- The following built-in function generates instructions to calculate and return the complex integral product of the complex conjugates of inputs a and b. The real and imaginary parts of the product are saturated to 16 bits to produce a `complex_int16` result.

```
complex_int16 conj_conj_cmul_i16(complex_int16 a, complex_int16 b);
```

- The following built-in function generates instructions to calculate and return the complex integral product of the inputs a and b. The real and imaginary parts of the product are saturated to 32 bits to produce a complex_int32 result.

  ```
  complex_int32 cmulw_i16(complex_int16 a, complex_int16 b);
  ```

- The following built-in function generates instructions to calculate and return the complex integral product of the input a and the complex conjugate of input b. The real and imaginary parts of the product are saturated to 32 bits to produce a complex_int32 result.

  ```
  complex_int32 conj_cmulw_i16(complex_int16 a, complex_int16 b);
  ```

- The following built-in function generates instructions to calculate and return the complex integral product of the complex conjugates of inputs a and b. The real and imaginary parts of the product are saturated to 32 bits to produce a complex_int32 result.

  ```
  complex_int32 conj_conj_cmulw_i16(complex_int16 a, complex_int16 b);
  ```

Changing the RND_MOD Bit

Three built-in functions (set_rnd_mod_biased, set_rnd_mod_unbiased, and restore_rnd_mod) provide a convenient way to change the state of the RND_MOD bit that controls whether the hardware performs biased or unbiased rounding. The builtins.h header file should be included to use these built-in functions.

- The following built-in function generates instructions to set the RND_BIT bit. This means that instructions that depend on the state of the RND_MOD bit will perform biased rounding. The previous state of the RND_MOD bit is returned.

  ```
  int set_rnd_mod_biased(void);
  ```

- The following built-in function generates instructions to unset the RND_BIT bit. This means that instructions that depend on the state of the RND_MOD bit perform unbiased rounding. The previous state of the RND_MOD bit is returned.

  ```
  int set_rnd_mod_unbiased(void);
  ```

- The following built-in function generates instructions to reset the RND_BIT bit to a previous value, which is passed into the function.

  ```
  void restore_rnd_mod(int);
  ```

The following example shows how to use these built-in functions.

```
#include <stdfix.h>
#include <builtins.h>

fract divide_biased(fract num, fract denom)
{
    fract rtn;
    int prev_rnd_mod = set_rnd_mod_biased();
    #pragma FX_ROUNDING_MODE BIASED;
    rtn = num / denom;
    restore_rnd_mod(prev_rnd_mod);
```

```
        return rtn;
}
```

Note that the pragma to set `FX_ROUNDING_MODE` is necessary due to the use of the `fract` type in the example. This pragma does not affect the state of the `RND_MOD` bit. See #pragma FX_ROUNDING_MODE {TRUNCATION|BIASED|UNBIASED} and Setting the Rounding Mode for further details.

Complex Operations in C++

Enabling the -full-cpplib switch ensures that complex operations adhere to the ISO/IEC 14882:2003 C++ standard. When using the abridged C++ library, the complex class is defined by the Analog Devices specific `<complex>` header file, and defines a template class for manipulating complex numbers. The standard arithmetic operators are overloaded, and there are `real()` and `imag()` methods for obtaining the relevant part of the complex number. C++

For example, the determinate and inverse of a `2x2` matrix of complex doubles may be computed using the following C++ function:

```
#include <complex>
using std::complex;

complexdouble> inverse2d(const complexdouble> mx[4],
                         complexdouble> mxinv[4])
{
   complexdouble> det = mx[0] * mx[3] - mx[2] * mx[1];

   if( (det.real() != 0.0) || (det.imag() != 0.0) ) {
      complexdouble> invdet = complexdouble>(1.0,0.0) / det;

      mxinv[0] =   invdet * mx[3];
      mxinv[1] = -(invdet * mx[1]);
      mxinv[2] = -(invdet * mx[2]);
      mxinv[3] =   invdet * mx[0];
   }
   return det;
}
```

By comparison, the equivalent function in C is:

```
#include <complex.h>

complex_double inverse2d(const complex_double mx[4],
                         complex_double mxinv[4])
{
   complex_double det;
   complex_double invdet;
   complex_double tmp;

   det = cmlt(mx[0],mx[3]);
   tmp = cmlt(mx[2],mx[1]);
   det = csub(det,tmp);
```

```
    if( (det.re != 0.0) || (det.im != 0.0) ) {
        invdet = cdiv((complex_double){1.0,0.0},det);

        mxinv[0] = cmlt(invdet,mx[3]);
        mxinv[1] = cmlt(invdet,mx[1]);
        mxinv[1].re = -mxinv[1].re;
        mxinv[1].im = -mxinv[1].im;
        mxinv[2] = cmlt(invdet,mx[2]);
        mxinv[2].re = -mxinv[2].re;
        mxinv[2].im = -mxinv[2].im;
        mxinv[3] = cmlt(invdet,mx[0]);
    }
    return det;
}
```

Packed 16-Bit Integer Built-In Functions

The compiler provides built-in functions that manipulate and perform basic arithmetic functions on two 16-bit integers packed into a single 32-bit type, `int2x16`. Use of the built-in functions produce optimal code sequences, using vectorized operations where possible. The types and operations are defined in the `i2x16.h` header file.

Composition and decomposition of the packed type are performed with the following functions:

```
int2x16 compose_i2x16(short _x, short _y);
short high_of_i2x16(int2x16 _x);
short low_of_i2x16(int2x16 _x);
```

The following functions perform vectorized arithmetic operations:

```
int2x16 abs_i2x16(int2x16 _x);
int2x16 add_i2x16(int2x16 _x, int2x16 _y);
int2x16 max_i2x16(int2x16 _x, int2x16 _y);
int2x16 min_i2x16(int2x16 _x, int2x16 _y);
int2x16 mult_i2x16(int2x16 _x, int2x16 _y);
int2x16 sub_i2x16(int2x16 _y, int2x16 _y);
```

The following functions perform vectorized shift operations:

```
int2x16 shl_i2x16(int2x16 _x, short _y);
            // arithmetic shift left (shifts right if second operand negative)
int2x16 shr_i2x16(int2x16 _x, short _y);
            // arithmetic shift right (shifts left if second operand negative)
int2x16 shll_i2x16(int2x16 _x, short _y);
            // logical shift left (shifts right if second operand negative)
int2x16 shrl_i2x16(int2x16 _x, short _y);
            // logical shift right (shifts left if second operand negative)
```

For completeness, the following functions perform the equivalent shift operations on scalar 16-bit inputs:

```
short shl_i1x16(short _x, short _y);
short shr_i1x16(short _x, short _y);
```

```
short shll_i1x16(short _x, short _y);
short shrl_i1x16(short _x, short _y);
```

The following function performs summation of the two packed components:

```
int sum_i2x16(int2x16 _x);
```

The following functions provide cross-wise multiplication:

```
long mult_ll_i2x16(int2x16 _x, int2x16 _y);
long mult_hl_i2x16(int2x16 _x, int2x16 _y);
long mult_lh_i2x16(int2x16 _x, int2x16 _y);
long mult_hh_i2x16(int2x16 _x, int2x16 _y);
```

The following function:

```
int2x16 add_on_sign(int2x16 _x, int2x16 _y);
```

is equivalent to the following operation on _x { a, b} and _y { c, d}, returning result {r, r}:

```
r = ((a < 0) ? -b : b) + ((c < 0) ? -d : d);
```

Division Functions

Two built-in functions (`divs` and `divq`) provide access to the "divide primitive" instructions:

```
#include <builtins.h>

int divs(int numerator, int denominator, int *aq);
int divq(int partialres, int denominator, int *aq);
```

The `divs()` and `divq()` built-in functions give access to the respective Blackfin instructions, DIVS and DIVQ, that are the foundation elements of a non-restoring, conditional, add-subtract, integer division algorithm.

The dividend (*numerator*) is a 32-bit value, and the divisor (*denominator*) is a 16-bit value; the high half of denominator is ignored. For details of the instructions, refer to ***DIVS, DIVQ (Divide Primitive)*** in the *Blackfin Processor Programming Reference* manual.

First, `divs()` initializes the processor's AQ flag and the quotient's sign bit (the initial value for *partialres*); successive uses of `divq()` generate a value bit for the quotient, producing a new *partialres*, and update the AQ flag. The *aq* parameter is used by the compiler to track the value of the AQ flag; `divs()` writes to **aq*, and each invocation of `divq()` updates **aq*. Typically, when optimizing, these reads and writes will be optimized away.

The following example uses the `divs()` and `divq()` primitives to implement a saturating, fractional division algorithm.

```
#include <builtins.h>
#include <fract.h>

fract16 saturating_fract_divide(fract16 nom, fract16 denom)
{
    int partialres = (int)nom;
    int divisor = (int)denom;
    fract16 rtn;
```

```
      int i;
      int aq;    /* initial value irrelevant */
      if (partialres == 0) {
         /* 0/anything gives 0 */
         rtn = 0;
   } else if (partialres >= divisor) {
         /* fract16 values have the range -1.0 = x  +1.0,  */
         /* so our result cannot be as high as 1.0.          */
         /* Therefore, for x/y, if x is larger than y, */
       /* saturate the result to positive maximum.    */
         rtn = 0x7fff;
   } else {
         /* nom is a 16-bit fractional value, so move */
         /* the 16 bits to the top of partialres.     */
         /* (promote fract16 to fract32) */
         partialres <<= 16;
         /* initialize sign bit and AQ, via divs(). */
         partialres = divs(partialres, divisor, &aq);

         /* Update each of the value bits of the partial result */
         /* and reset AQ via divq().                            */
         for (i=0; i<15; i++) {
            partialres = divq(partialres, divisor, &aq);
         }
         rtn = (fract16) partialres;
   }
   return rtn;
}
```

Full-Precision Accumulator Built-In Functions

The compiler provides built-in functions to take advantage of the full 40-bit precision of the accumulator registers.

Fractional Dot Product Implemented with Accumulator Built-Ins

This example shows a dot product that is guaranteed to accumulate in 40-bits and to saturate the final sum to 32-bits.

```
#include <builtins.h>

fract32 dot(fract16 a[], fract16 b[], int n)
{
   int i;
   acc40 sum = 0;
   for (i = 0; i < n; ++i)
      sum = A_mac(sum, a[i], b[i]);
   return A_mad(sum);
}
```

Fractional Dot Product Implemented with Accumulator Pair Built-Ins on the Blackfin+ series of processors

On the Blackfin+ series of processors, built-in functions are also provided to take advantage of the full 72-bit precision of an accumulator register pair. 72-bit values are held in variables of the `acc72` type (defined in the `builtins.h` header file). This is a 128-bit container type which can only be manipulated with built-in functions. You cannot perform native arithmetic on variables of this type. This example shows a fractional dot product that is guaranteed to accumulate in 72-bits and to saturate and round the final sum to fit in 32-bits.

```
#include <builtins.h>

fract32 dot(fract32 a[], fract32 b[], int n)
{
   int i;   acc72 sum = compose_a01(0, 0);
   for (i = 0; i < n; ++i)
      sum = A_macw(sum, a[i], b[i]);
   return A_madw(sum);
}
```

Complex Fractional Dot Product Implemented with Accumulator Built-Ins on the Blackfin+ series of processors

The Blackfin+ series of processors also has instructions that perform complex multiplication, leaving the real and imaginary parts of the result in the two accumulators. Built-in functions are provided to take advantage of the full 40-bit precision of the two accumulator results. This example shows a complex fractional dot product that is guaranteed to accumulate the real and imaginary parts in 40-bits and to saturate and round the final values to 32-bits.

```
#include <builtins.h>

complex_fract16 dot(complex_fract16 a[], complex_fract16 b[], int n)
{
   int i;
   complex_fract16 sum;
   acc40 sum_re = 0;
   acc40 sum_im = 0;
   for (i = 0; i < n; ++i)
       sum_re = A_cmac(sum_re, sum_im, a[i], b[i], &sum_im);
   sum.re = A_madh(sum_re);
   sum.im = A_madh(sum_im);
   return sum;
}
```

Accumulator Built-In Function Prototypes

The following tables list all the full-precision accumulator built-in functions with their characteristic instruction. Each function implements the same computation as this characteristic instruction, but the compiler may generate an alternative instruction sequence to do so.

NOTE: The results of the functions marked with a dagger (†) in the *Accumulator Built-In Functions* table are affected by the setting of the RND_MOD bit in the ASTAT register. See the *Blackfin Processor Programming Reference* manual for details.

Table 2-30: Accumulator Built-In Functions

Function	*Instruction*
`acc40 A_mult(fract16, fract16);`	`An = Dx.lh * Dy.lh`
`acc40 A_mult_FU(fract16, fract16);`	`An = Dx.lh * Dy.lh (FU)`
`acc40 A_mult_M(fract16, fract16);`	`A1 = Dx.lh * Dy.lh (M)`
`acc40 A_mult_IS(short, short);`	`An = Dx.lh * Dy.lh (IS)`
`acc40 A_mult_MIS(short, unsigned short);`	`A1 = Dx.lh * Dy.lh (M,IS)`
`acc40 A_mac(acc40,fract16, fract16);`	`An += Dx.lh * Dy.lh`
`acc40 A_mac_FU(acc40,fract16, fract16);`	`An += Dx.lh * Dy.lh (FU)`
`acc40 A_mac_M(acc40,fract16, fract16);`	`A1 += Dx.lh * Dy.lh (M)`
`acc40 A_mac_IS(acc40,short, short);`	`An += Dx.lh * Dy.lh (IS)`
`acc40 A_mac_MIS(acc40,short, unsigned short);`	`A1 += Dx.lh * Dy.lh (M,IS)`
`acc40 A_msu(acc40,fract16, fract16);`	`An -= Dx.lh * Dy.lh`
`acc40 A_msu_FU(acc40,fract16, fract16);`	`An -= Dx.lh * Dy.lh (FU)`
`acc40 A_msu_M(acc40,fract16, fract16);`	`A1 -= Dx.lh * Dy.lh (M)`
`acc40 A_msu_IS(acc40,short, short);`	`An -= Dx.lh * Dy.lh (IS)`
`acc40 A_msu_MIS(acc40,short, unsigned short);`	`A1 -= Dx.lh * Dy.lh (M,IS)`
`int A_eq(acc40, acc40);`	`CC = A0 == A1`
`int A_lt(acc40, acc40);`	`CC = A0 A1`
`int A_le(acc40, acc40);`	`CC = A0 = A1`
`acc40 A_add(acc40, acc40);`	`A0 += A1`
`acc40 A_sub(acc40, acc40);`	`A0 -= A1`
`acc40 A_neg(acc40);`	`An = -An`
`acc40 A_abs(acc40);`	`An = ABS An`
`int A_bitmux_ASR(int, int, acc40, int*, acc40*);`	`BITMUX(Dx, Dy, A0) (ASR)`
`int A_bitmux_ASL(int, int, acc40, int*, acc40*);`	`BITMUX(Dx, Dy, A0) (ASL)`
`short A_bxorshift_mask32(acc40, int, int*);`	`Dn.L = CC = BXORSHIFT(A0, Dx)`
`short A_bxor_mask32(acc40, int, int*);`	`Dn.L = CC = BXOR(A0, Dx)`
`acc40 A_bxorshift_mask40(acc40, acc40, int);`	`A0 = BXORSHIFT(A0, A1, CC);`
`short A_bxor_mask40(acc40, acc40, int, int*);`	`Dn.L = CC = BXOR(A0, A1, CC);`
`short A_signbits(acc40);`	`Dx.L = SIGNBITS An;`

Table 2-30: Accumulator Built-In Functions (Continued)

Function	Instruction

`acc40 A_ashift(acc40, short);`
(See Note 1)
	`An = ASHIFT An BY Dx.L`
	`An = An >>> uimm5`
	`An = An uimm5`

`acc40 A_lshift(acc40, short);`
(See Note 1)
	`An = LSHIFT An BY Dx.L`
	`An = An >> uimm5`
	`An = An uimm5`

`acc40 A_sat(acc40);`	`An = An (S)`
`fract32 A_mad(acc40);`	`Dn = An`
`fract32 A_mad_FU(acc40);`	`Dn = An (FU)`
`fract32 A_mad_S2RND(acc40);`	`Dn = An (S2RND)`
`int A_mad_ISS2(acc40);`	`Dn = An (ISS2)`
`fract16 A_madh(acc40);` (See Note 2)	`Dn.lh = An`
`fract16 A_madh_FU(acc40);` (See Note 2)	`Dn.lh = An (FU)`
`short A_madh_IS(acc40);`	`Dn.lh = An (IS)`
`unsigned short A_madh_IU(acc40);`	`Dn.lh = An (IU)`
`fract16 A_madh_T(acc40);`	`Dn.lh = An (T)`
`fract16 A_madh_TFU(acc40);`	`Dn.lh = An (TFU)`
`fract16 A_madh_S2RND(acc40);` (See Note 2)	`Dn.lh = An (S2RND)`
`short A_madh_ISS2(acc40);`	`Dn.lh = An (ISS2)`
`short A_madh_IH(acc40);` (See Note 2)	`Dn.lh = An (IH)`

NOTE: 1. These functions return their first operand An shifted left by Dx.L places if Dx.L is positive, or shifted right by ABS(Dx.L) places if Dx.L is negative. See the *Blackfin Processor Programming Reference* manual for details.

NOTE: 2. The results of these functions are affected by the setting of the RND_MOD bit in the ASTAT register. See the *Blackfin Processor Programming Reference* manual for details.

Table 2-31: Blackfin+ Accumulator Pair Built-in Functions for 32-bit Multiplies.

Function	Instruction
`acc72 A_multw(fract32, fract32);`	`A1:0 = Dx * Dy`

Table 2-31: Blackfin+ Accumulator Pair Built-in Functions for 32-bit Multiplies. (Continued)

Function	Instruction
`acc72 A_macw(acc72, fract32, fract32);`	`A1:0 += Dx * Dy`
`acc72 A_msuw(acc72, fract32, fract32);`	`A1:0 -= Dx * Dy`
`acc72 A_multw_FU(fract32, fract32);`	`A1:0 = Dx * Dy (FU)`
`acc72 A_macw_FU(acc72, fract32, fract32);`	`A1:0 += Dx * Dy (FU)`
`acc72 A_msuw_FU(acc72, fract32, fract32);`	`A1:0 -= Dx * Dy (FU)`
`acc72 A_multw_M(fract32, fract32);`	`A1:0 = Dx * Dy (M)`
`acc72 A_macw_M(acc72, fract32, fract32);`	`A1:0 += Dx * Dy (M)`
`acc72 A_msuw_M(acc72, fract32, fract32);`	`A1:0 -= Dx * Dy (M)`
`acc72 A_multw_IS(int, int);`	`A1:0 = Dx * Dy (IS)`
`acc72 A_macw_IS(acc72, int, int);`	`A1:0 += Dx * Dy (IS)`
`acc72 A_msuw_IS(acc72, int, int);`	`A1:0 -= Dx * Dy (IS)`
`acc72 A_multw_ISNS(int, int);`	`A1:0 = Dx * Dy (IS, NS)`
`acc72 A_macw_ISNS(acc72, int, int);`	`A1:0 += Dx * Dy (IS, NS)`
`acc72 A_msuw_ISNS(acc72, int, int);`	`A1:0 -= Dx * Dy (IS, NS)`
`acc72 A_multw_IU(unsigned int, unsigned int);`	`A1:0 = Dx * Dy (IU)`
`acc72 A_macw_IU(acc72, unsigned int, unsigned int);`	`A1:0 += Dx * Dy (IU)`
`acc72 A_msuw_IU(acc72, unsigned int, unsigned int);`	`A1:0 -= Dx * Dy (IU)`
`acc72 A_multw_IUNS(unsigned int, unsigned int);`	`A1:0 = Dx * Dy (IU, NS)`
`acc72 A_macw_IUNS(acc72, unsigned int, unsigned int);`	`A1:0 += Dx * Dy (IU, NS)`
`acc72 A_msuw_IUNS(acc72, unsigned int, unsigned int);`	`A1:0 -= Dx * Dy (IU, NS)`
`acc72 A_multw_MIS(int, unsigned int);`	`A1:0 = Dx * Dy (M, IS)`
`acc72 A_macw_MIS(acc72, int, unsigned int);`	`A1:0 += Dx * Dy (M, IS)`
`acc72 A_msuw_MIS(acc72, int, unsigned int);`	`A1:0 -= Dx * Dy (M, IS)`
`acc72 A_multw_MISNS(int, unsigned int);`	`A1:0 = Dx * Dy (M, IS, NS)`
`acc72 A_macw_MISNS(acc72, int, unsigned int);`	`A1:0 += Dx * Dy (M, IS, NS)`
`acc72 A_msuw_MISNS(acc72, int, unsigned int);`	`A1:0 -= Dx * Dy (M, IS, NS)`
`fract32 A_madw(acc72);`	`Dx = A1:0`
`fract32 A_madw_T(acc72);`	`Dx = A1:0 (T)`
`fract32 A_madw_FU(acc72);`	`Dx = A1:0 (FU)`
`fract32 A_madw_TFU(acc72);`	`Dx = A1:0 (TFU)`
`fract32 A_madw_M(acc72);`	`Dx = A1:0 (M)`

Table 2-31: Blackfin+ Accumulator Pair Built-in Functions for 32-bit Multiplies. (Continued)

Function	Instruction
`fract32 A_madw_MT(acc72);`	`Dx = A1:0 (M, T)`
`int A_madw_IS(acc72);`	`Dx = A1:0 (IS)`
`int A_madw_ISNS(acc72);`	`Dx = A1:0 (IS, NS)`
`unsigned int A_madw_IU(acc72);`	`Dx = A1:0 (IU)`
`unsigned int A_madw_IUNS(acc72);`	`Dx = A1:0 (IU, NS)`
`int A_madw_MIS(acc72);`	`Dx = A1:0 (M, IS)`
`int A_madw_MISNS(acc72);`	`Dx = A1:0 (M, IS, NS)`
`fract64 A_madww(acc72);`	`Dx+1:x = A1:0`
`fract64 A_madww_FU(acc72);`	`Dx+1:x = A1:0 (FU)`
`fract64 A_madww_M(acc72);`	`Dx+1:x = A1:0 (M)`
`long long A_madww_IS(acc72);`	`Dx+1:x = A1:0 (IS)`
`long long A_madww_ISNS(acc72);`	`Dx+1:x = A1:0 (IS, NS)`
`unsigned long long A_madww_IU(acc72);`	`Dx+1:x = A1:0 (IU)`
`unsigned long long A_madww_IUNS(acc72);`	`Dx+1:x = A1:0 (IU, NS)`
`long long A_madww_MIS(acc72);`	`Dx+1:x = A1:0 (M, IS)`
`long long A_madww_MISNS(acc72);`	`Dx+1:x = A1:0 (M, IS, NS)`
`acc40 extract_a0(acc72);`	access A0 part of `acc72` accumulator pair
`acc40 extract_a1(acc72);`	access A1 part of `acc72` accumulator pair
`acc72 compose_a01(acc40 a0, acc40 a1);`	compose `acc72` result from A0 and A1 values

Table 2-32: Blackfin+ Accumulator Built-In Functions for Complex Multiplies. (In each case the returned result is the real part.)

Function	Instruction
`acc40 A_cmul(complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 = CMUL(Dx, Dy)`
`acc40 A_cmac(acc40 re, acc40 im, complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 += CMUL(Dx, Dy)`
`acc40 A_cmsu(acc40 re, acc40 im, complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 -= CMUL(Dx, Dy)`
`acc40 A_conj_cmul(complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 = CMUL(Dx, Dy*)`
`acc40 A_conj_cmac(acc40 re, acc40 im, complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 += CMUL(Dx, Dy*)`
`acc40 A_conj_cmsu(acc40 re, acc40 im, complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 -= CMUL(Dx, Dy*)`

Table 2-32: Blackfin+ Accumulator Built-In Functions for Complex Multiplies. (In each case the returned result is the real part.) (Continued)

Function	*Instruction*
`acc40 A_conj_conj_cmul(complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 = CMUL(Dx*, Dy*)`
`acc40 A_conj_conj_cmac(acc40 re, acc40 im, complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 += CMUL(Dx*, Dy*)`
`acc40 A_conj_conj_cmsu(acc40 re, acc40 im, complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 -= CMUL(Dx*, Dy*)`
`acc40 A_cmul_IS(complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 = CMUL(Dx, Dy) (IS)`
`acc40 A_cmac_IS(acc40 re, acc40 im, complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 += CMUL(Dx, Dy) (IS)`
`acc40 A_cmsu_IS(acc40 re, acc40 im, complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 -= CMUL(Dx, Dy) (IS)`
`acc40 A_conj_cmul_IS(complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 = CMUL(Dx, Dy*) (IS)`
`acc40 A_conj_cmac_IS(acc40 re, acc40 im, complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 += CMUL(Dx, Dy*) (IS)`
`acc40 A_conj_cmsu_IS(acc40 re, acc40 im, complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 -= CMUL(Dx, Dy*) (IS)`
`acc40 A_conj_conj_cmul_IS(complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 = CMUL(Dx*, Dy*) (IS)`
`acc40 A_conj_conj_cmac_IS(acc40 re, acc40 im, complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 += CMUL(Dx*, Dy*) (IS)`
`acc40 A_conj_conj_cmsu_IS(acc40 re, acc40 im, complex_fract16, complex_fract16, acc40 *im_out);`	`A1:0 -= CMUL(Dx*, Dy*) (IS)`

Table 2-33: Types Used in Accumulator Built-In Functions

C Type	*Usage*
`acc40`	Any value in an accumulator. This is a signed 64-bit integer containing the 40-bit accumulator value. The most significant 24 bits are ignored by these built-in functions. 40-bit accumulator values are sign-extended to 64 bits when moving values from accumulator registers to other registers or memory.
`acc72`	Any value in the accumulator pair, `A1:0`. This is a signed 128-bit integer containing the 72-bit accumulator pair value. The bottom 32 bits of the A0 register are held in bits [0,31] of the 128-bit value. The 40 bits of the A1 register are held in bits [64,83] of the 128-bit value. The most significant 24 bits, and bits [32,63] are ignored by these built-in functions. This type is only used in built-ins defined for the Blackfin+ series of processors.
`fract32`	32-bit signed or unsigned fractional value
`fract16`	16-bit signed or unsigned fractional value
`int`	32-bit signed integer value

Table 2-33: Types Used in Accumulator Built-In Functions (Continued)

C Type	Usage
unsigned	32-bit unsigned integer value
short	16-bit signed integer value
unsigned short	16-bit unsigned integer value
Dx, Dy, Dn	Data registers (R0 ... R7)
Dx+1:x	Data register pairs (R1:0, R3:2, R5:4, R7:6)
lh	A low-half specifier (.L) or a high-half specifier (.H)
An	Accumulator registers (A0 or A1)

Accumulator Built-In Functions and the Optimizer

The compiler usually generates an accumulator instruction for each call to an accumulator built-in function, but it does not map acc40-typed variables to accumulator registers unless optimization is enabled. See the enable optimizations switch (-O[0|1]).

Other circumstances may impact the efficiency of the generated code; for example, the Blackfin processor has two 40-bit accumulator registers, so C code that has more than two acc40 variables in use at the same time will require some inefficient shuffling of values in and out of the accumulators to perform the calculation.

The accumulator data type acc40 is a signed 64-bit integral type, so arithmetic operators can be used with variables of this type. However, this is not equivalent to using the accumulator intrinsics and usually translates to expensive 64-bit arithmetic, which may offset any performance benefit of using an accumulator. In addition, the acc40 type should not be confused with the native fixed-point type accum available through the stdfix.h header file.

Since the acc40 type is a signed 64-bit integral type, constants used to initialize it are interpreted as 64 bits in size. For example, the code:

```
#include <builtins.h>
acc40 acc = 0x80000000;
```

results in the accumulator register being initialized to 0x0080000000, not 0xff80000000.

When optimization is enabled, the compiler may also use accumulator registers to implement short multiplication and int addition operations. This use of a 40-bit accumulator to implement 32-bit addition will produce the same results as long as the 32-bit operation would not have overflowed.

Comparison of Two Dot Products

Consequently, the two versions of dot product in this example may translate to the same assembly code depending on compilation options, but only the version that uses the A_mac_IS built-in function is guaranteed to compute the same result as an assembly function which uses an accumulator register, for all possible inputs and with any compiler option. If your computations are at risk of overflow and you want to be certain that saturation does not occur, consider using the -no-saturation switch. This switch prevents the use of accumulator registers for addition operations but at the expense of reduced performance.

```
#include <builtins.h>

/* may accumulate in 40 bits with optimization, ** but not guaranteed. */
int dot32(short a[], short b[], int n)
{
   int i;      int sum = 0;
   for (i = 0; i < n; ++i)          sum += a[i] * b[i];
   return sum;
}
/* guaranteed to accumulate in 40 bits */
int dot40(short a[], short b[], int n)
{
   int i;      acc40 sum = 0;
   for (i = 0; i < n; ++i)          sum = A_mac_IS(sum, a[i], b[i]);
   return (int)sum;
}
```

Viterbi History and Decoding Functions

Four built-in functions provide the selection function of a Viterbi decoder. Specifically, these four functions provide the maximum value selection and history update parts. The functions use the A0 accumulator to maintain the history value. (The accumulator register maintains the history values by shifting the previous value along one place and setting a bit to indicate the result of the current iteration's selection.)

To use the Viterbi functions, you must include ccblkfn.h in the source modules in which they are used. Failure to do so leads to errors at compile-time.

The four Viterbi functions allow for left- or right-shifting (setting the least or most significant bit, accordingly) and for 1x16 or 2x16 operands.

The first two functions provide left- and right-shifting operations for single 16-bit input operands:

```
short lvitmax1x16(int value, int oldhist, int *newhist);
short rvitmax1x16(int value, int oldhist, int *newhist);
```

lvitmax1x16() and rvitmax1x16() perform selection-and-update operations for two 16-bit operands, which are in the high and low halves of value. The oldhist operand contains the history value from the preceding iteration. The short value returned contains the selection result, and the pointer newhist contains the history state after the operation.

The returned value is set to contain the largest half of value. The newhist operand is set to contain the oldhist value, shifted one place (left for lvitmax, right for rvitmax), and with one bit (LSB for lvitmax, MSB for rvitmax) set to 1 if the high half was selected; 0 otherwise.

The next two Viterbi functions provide left- and right-shifting operations for pairs of 16-bit input operands. The functions are:

```
int lvitmax2x16(int val_x, int val_y, int oldhist, int *newhist);
int rvitmax2x16(int val_x, int val_y, int oldhist, int *newhist);
```

The two functions, `lvitmax2x16()` and `rvitmax2x16()`, perform two selection-and-update operations. Each of the `val_x` and `val_y` input expressions contain two 16-bit operands. A selection operation is performed on the two 16-bit operands in `val_x`, and another selection operation is performed on the two 16-bit operands in `val_y`. The `oldhist` value is shifted and updated into `newhist`, as described above.

However, in this example, `oldhist` is shifted two places, and two bits are set. The history value is shifted one place, and a bit is set to indicate the result of the `val_x` selection operation. Then, the history value is shifted a second place, and another bit is set to indicate the result for the `val_y` selection operation.

The selected value from `val_x` is stored in the low half of the returned value, and the selected value from `val_y` is stored in the high half.

Search Built-in Functions

The compiler provides several built-in functions for locating the largest or smallest 16-bit signed values in an array, using a loop. Each version of the search built-in function has the following signature:

```
int2x16 *search_op(int2x16 cmp_vals,
                   int2x16 *cmp_ptr,
                   int2x16 *prev_hi_ptr,
                   int2x16 *prev_lo_ptr,
                   short prev_hi,
                   short prev_lo,
                   int2x16 **new_lo_ptr,
                   short *new_hi,
                   short *new_lo);
```

The available search functions are listed in the *Built-in Search Functions* table. Each invocation of a search function compares two values from the array against current best solutions, updating those partial results if appropriate. If a value being tested is better than the current solution, the function also saves the current pointer.

Upon completion of the search process, the function will have identified two parallel sets of results, one for the values in the low half of the int2x16 value, and one for the values in the high half. Each set of results contain the best solution identified (i.e. the largest or smallest value) and the corresponding pointer value.

The function returns the new pointer value for the low half comparison, and passes the new pointer value for the high half comparison back via `new_lo_ptr`. The new partial results are returned in `new_hi` and `new_lo`.

Table 2-34: Built-in Search Functions

Function Name	*Operation*
`search_gt`	`new = (cmp > prev)? cmp : prev` `new_ptr = (cmp > prev)? cmp_ptr : prev_ptr`
`search_ge`	`new = (cmp >= prev)? cmp : prev` `new_ptr = (cmp >= prev)? cmp_ptr : prev_ptr`
`search_lt`	`new = (cmp < prev)? cmp : prev` `new_ptr = (cmp < prev)? cmp_ptr : prev_ptr`

Table 2-34: Built-in Search Functions (Continued)

Function Name	*Operation*
`search_le`	`new = (cmp <= prev)? cmp : prev` `new_ptr = (cmp <= prev)? cmp_ptr : prev_ptr`

Circular Buffer Built-In Functions

The C/C++ compiler provides built-in functions that use the Blackfin processor's circular buffer mechanisms. These functions provide automatic circular buffer generation, circular indexing, and circular pointer references.

Automatic Circular Buffer Generation

If optimization is enabled, the compiler automatically attempts to use circular buffer mechanisms where appropriate. For example,

```
void func(int *array,int n,int incr)
{
   int i;
   for (i = 0; i < n; i++)
      array [ i % 10 ] += incr;
}
```

The compiler recognizes that the `[i % 10]` expression is a circular reference, and uses a circular buffer if possible. There are cases where the compiler is unable to verify that the memory access is always within the bounds of the buffer. The compiler is conservative in such cases, and does not generate circular buffer accesses.

Even in such cases, the compiler can be instructed to generate circular buffer accesses by specifying the `-force-circbuf` switch. For more information, see -force-circbuf.

Explicit Circular Buffer Generation

The compiler also provides built-in functions that can explicitly generate circular buffer accesses, subject to available hardware resources. The built-in functions provide circular indexing and circular pointer references. Both built-in functions are defined in the `ccblkfn.h` header file.

Circular Buffer Increment of an Index

The following operation performs a circular buffer increment of an index.

```
ptrdiff_t circindex(ptrdiff_t ptr, ptrdiff_t incr, size_t len);
```

The operation is equivalent to:

```
index += incr;
if (index  0)
      index += len;
else if (index >= len)
      index -= len;
```

An example of this built-in function is:

```
#include <ccblkfn.h>
```

```
void func(int *array, int n, int incr, int len)
{
   int i, idx = 0;

   for (i = 0; i < n; i++) {
       array[idx] += incr;
       idx = circindex(idx, incr, len);
   }
}
```

NOTE: For correct operation, the increment should not exceed the buffer length.

Circular Buffer Increment of a Pointer

The following operation performs a circular buffer increment of a pointer.

```
void *circptr(const void *ptr, ptrdiff_t incr,
             const void *base, size_t buflen);
```

Both `incr` and `buflen` are specified in bytes, since the operation deals in void pointers.

The operation is equivalent to:

```
ptr += incr;
if (ptr < base)
   ptr += buflen;
else if (ptr >= (base+buflen))
   ptr -= buflen;
```

An example of this built-in function is:

```
#include <ccblkfn.h>

void func(int *array, int n, int incr, int len)
{
   int i, idx = 0;
   int *ptr = array;

   // scale increment and length by size
   // of item pointed to.
   incr *= sizeof(*ptr);
   len *= sizeof(*ptr);

   for (i = 0; i < n; i++) {
       *ptr += incr;
       ptr = circptr(ptr, incr, array, len);
   }
}
```

NOTE: For correct operation, the increment should not exceed the buffer length.

Endian-Swapping Intrinsics

The following two intrinsics are available for changing data from big-endian to little-endian, or vice versa.

```
#include <ccblkfn.h>

int byteswap4(int);
short byteswap2(short);
```

For example, `byteswap2(0x1234)` returns `0x3412`.

Since Blackfin processors use a little-endian architecture, these intrinsics are useful when communicating with big-endian devices, or when using a protocol that requires big-endian format. For example,

```
struct bige_buffer {
    int len;
    char data[MAXLEN];
} buf;
int i, len;
buf = get_next_buffer();
len = byteswap4(buf.len);
for (i = 0; i < len; i++)
process_byte(buf.data[i]);
```

System Built-In Functions

The following built-in functions allow access to system facilities on Blackfin processors. The functions are defined in the `ccblkfn.h` header file. Include the `ccblkfn.h` file before using these functions. Failure to do so leads to unresolved symbols at link-time.

Stack Space Allocation

```
void *alloca(unsigned);
```

This function allocates the requested number of bytes on the local stack, and returns a pointer to the start of the buffer. The space is freed when the current function exits.

The compiler supports this function via `__builtin_alloca()`.

System Register Values

```
unsigned int sysreg_read(int reg);
void sysreg_write(int reg, unsigned int val);
unsigned long long sysreg_read64(int reg);
void sysreg_write64(int reg,unsigned long long val);
```

These functions get (read) or set (write) the value of a system register. In all cases, `reg` is a constant from the file `<sysreg.h>`.

IMASK Values

```
unsigned cli(void);
void sti(unsigned mask);
```

The `cli()` function retrieves the old value of `IMASK`, and disables interrupts by setting `IMASK` to all zeros. The `sti()` function installs a new value into `IMASK`, enabling the interrupt system according to the new mask stored.

Interrupts and Exceptions

```
void raise_intr(int);
void excpt(int);
```

These two functions raise interrupts and exceptions, respectively. In both cases, the parameter supplied must be an integer literal value.

Idle Mode

```
void idle(void);
```

Places the processor in idle mode.

On the Blackfin+ series of processors, the built-in function:

```
void sti_idle(unsigned int a);
```

installs a new value into `IMASK`, enabling the interrupt system according to the new mask stored, and then enters idle mode. This is equivalent to the `STI IDLE` hardware instruction.

Synchronization

```
void csync(void);
void ssync(void);
```

These two functions provide synchronization. The `csync()` function is a core-only synchronization - it flushes the pipeline and store buffers. The `ssync()` function is a system synchronization, and also waits for an `ACK` instruction from the system bus.

Cache Built-In Functions

The following built-in functions can be used to control the instruction and data caches.

flush

```
void __builtin_flush(void * __a);
```

When compiled, this built-in function is replaced by the assembly:

```
FLUSH[Preg]; // Preg is loaded with the address __a
```

The `__builtin_flush` (data cache line flush) function causes the data cache to synchronize the cache line associated with the specified address with higher levels of memory. If the cached data line is dirty, the instruction writes the line out and marks the line clean in the data cache. If the specified data cache line is already clean or does not exist, the instruction functions like a `NOP`.

flushinv

```
void __builtin_flushinv(void * __a);
```

When compiled, this built-in function is replaced by the assembly:

```
FLUSHINV[Preg]; // Preg is loaded with the address __a
```

The __builtin_flushinv (data cache line flush and invalidate) function causes the data cache to perform the same function as flush and then invalidate the specified line in the cache. If the line is in the cache and dirty, the cache line is first written out. The Valid bit in the cache line is then cleared. If the line is not in the cache, flushinv functions like a NOP.

flushinvmodup

```
void * __builtin_flushinvmodup(void * __a);
```

When compiled, this built-in function is replaced by the assembly:
```
FLUSHINV[Preg++]; // Preg is loaded with the address __a
```

The __builtin_flushinvmodup functions exactly the same way as flushinv; however, the specified address is post-incremented by the size of a cache block (for example, 32 bytes) and then returned.

flushmodup

```
void * __builtin_flushmodup(void * __a);
```

When compiled, this built-in function will be replaced by the assembly:
```
FLUSH[Preg++]; // Preg is loaded with the address __a
```

The __builtin_flushmodup functions exactly the same way as flush; however, the specified address is post-incremented by the size of a cache block (for example, 32 bytes) and then returned.

iflush

```
void * __builtin_iflush(void * __a);
```

When compiled, this built-in function is replaced by the assembly:
```
IFLUSH[Preg]; // Preg is loaded with the address __a
```

The __builtin_iflush (instruction cache flush) function causes the instruction cache to invalidate the cache line associated with the address specified. The instruction cache contains no dirty bit. Consequently, the contents of the instruction cache are never flushed to higher levels.

iflushmodup

```
void * __builtin_iflushmodup(void * __a);
```

When compiled, this built-in function is replaced by the assembly:
```
IFLUSH[Preg++]; // Preg is loaded with the address __a
```

The __builtin_iflushmodup functions exactly the same way as iflush; however, the specified address is post-incremented by the size of a cache block (for example, 32 bytes) and then returned.

prefetch

```
void * __builtin_prefetch(void * __a);
```

When compiled, this built-in function is replaced by the assembly:

```
PREFETCH[Preg]; // Preg is loaded with the address __a
```

The `__builtin_prefetch` (data cache prefetch) function causes the data cache to prefetch the cache line that is associated with the specified address. The operation causes the line to be fetched if it is not currently in the data cache and if the address is cacheable. If the line is already in the cache or if the cache is already fetching a line, `prefetch` performs like a `NOP`.

prefetchmodup

```
void * __builtin_prefetchmodup(void * __a);
```

When compiled, this built-in function is replaced by the assembly:

```
PREFETCH[Preg++]; // Preg is loaded with the address __a
```

The `__builtin_prefetchmodup` functions exactly the same way as prefetch; however, the specified address is post-incremented by the size of a cache block (for example, 32 bytes) and then returned.

Compiler Performance Built-In Functions

The compiler performance built-in functions do not have any effect on the functional behavior of compiled code. Instead, they provide the compiler with additional information about the code being compiled, allowing the compiler to generate more efficient code. The facilities are:

- Expected behavior, which allows you to tell the compiler which way a condition is most likely to be resolved.

- Known values, which allows you to tell the compiler about the values that your variables will have at certain points in the program.

Expected Behavior

The `expected_true` and `expected_false` functions provide the compiler with information about the expected behavior of the program. You can use these built-in functions to tell the compiler which parts of the program are most likely to be executed; the compiler then can arrange for the most common cases to be those that execute most efficiently.

```
#include <ccblkfn.h>
int expected_true(int cond);
int expected_false(int cond);
```

For example, consider the code:

```
extern int func(int);
int example(int call_the_function, int value)
{
   int r = 0;
   if (call_the_function)
      r = func(value);
   return r;
}
```

If you expect that parameter `call_the_function` to be true in the majority of cases, you can write the function in the following manner:

```
extern int func(int);
int example(int call_the_function, int value)
{
    int r = 0;
    if (expected_true(call_the_function))        // indicate most likely true
        r = func(value);
    return r;
}
```

This indicates to the compiler that you expect `call_the_function` to be true in most cases, so the compiler arranges for the default case to be to call function `func()`.

On the other hand, if you write the function as follows, the compiler arranges the generated code to default to the opposite case, of not calling function `func()`.

```
extern int func(int);
int example(int call_the_function, int value)
{
    int r = 0;
    if (expected_false(call_the_function))        // indicate most likely false
        r = func(value);
    return r;
}
```

These built-in functions do not change the operation of the generated code, which will still evaluate the boolean expression as normal. Instead, they indicate to the compiler which flow of control is most likely, helping the compiler to ensure that the most commonly-executed path is the one that uses the most efficient instruction sequence.

The expected_true and expected_false built-in functions take effect only when optimization is enabled in the compiler. They are supported in conditional expressions only.

Known Values

The `__builtin_assert()` function provides the compiler with information about the values of variables which it may not be able to deduce from the context. For example, consider the code:

```
int example(int value, int loop_count)
{
    int r = 0;
    int i;
        for (i = 0; i < loop_count; i++) {
            r += value;
    }
    return r;
}
```

The compiler has no way of knowing what values may be passed to the function. If you know that the loop count will always be greater than four, you can allow the optimizer to make use of that knowledge using `__builtin_assert()`.

```
int example(int value, int loop_count)
{
   int r = 0;
   int i;
   __builtin_assert(loop_count > 4);
     for (i = 0; i < loop_count; i++) {
         r += value;     }
   return r;
}
```

The optimizer can now omit the jump over the loop body it would otherwise have to emit to cover `loop_count == 0`. In more complicated code, further optimizations may be possible when bounds for variables are known.

Video Operation Built-In Functions

The C/C++ compiler provides built-in functions for using the Blackfin processor's video pixel operations. Include the `video.h` header file before using these functions.

Some video operation built-in functions take an 8-byte sequence of data, and select from it a sequence of four bytes to use as input. The operation selects the four bytes at an offset of 0, 1, 2, or 3 bytes from lowest byte of the 8-byte sequence, depending on the value of a pointer parameter. Where reverse variants of the operations exist (the operation name is suffixed by "`r`"), the two 4-byte halves of the 8-byte sequence are accessed in reverse order.

Where a video operation generates more than one result, the operation may be implemented by more than one built-in function. In these cases, macros are provided to generate the appropriate built-in calls.

For further information regarding the underlying Blackfin processor instructions that implement the video operations, refer to the *Blackfin Processor Programming Reference.*

Function Prototypes

Align Operations

```
int align8(int src1, int src2);    /* 1 byte offset */

int align16(int src1, int src2);   /* 2 byte offset */

int align24(int src1, int src2);   /* 3 byte offset */
```

These three operations treat their two inputs as a single 8-byte sequence, and extract a specific 4-byte sequence from it, starting at offset 1, 2, or 3 bytes, as shown.

Packing Operations

```
int bytepack(int src1, int src2);
```

This operation treats its two inputs as four 16-bit values, and packs each 16-bit value into an 8-bit value in the result. Effectively, it converts an array of four `shorts` to an array of four `chars`.

```
long long compose_i64(int low, int high);
```

This operation produces a 64-bit value from the two 32-bit values provided as input and can be used to efficiently generate a `long long` type that is needed for many of the following operations.

Misaligned Loads

```
int loadbytes(int *ptr);
```

This operation is used to load a 4-byte sequence from memory using `ptr` as the address, where `ptr` may be misaligned. The actual data retrieved is aligned by masking off the bottom two bits of `ptr`, where `ptr` is intended to select bytes from input operands in subsequent operations. Misaligned read exceptions are prevented from occurring. Note that, for Blackfin+ processors such as the ADSP-BF7xx processors, the architecture supports misaligned data accesses (although such accesses may incur a performance penalty to handle misalignment).

Unpacking

```
byteunpack(long long src, char *ptr, int dst1, int dst2);

byteunpackr(long long src, char *ptr, int dst1, int dst2);
```

These macros provide the unpacking operations, where `PTR` selects four bytes from the eight-byte sequence in `SRC`. Each of the four bytes is expanded to a 16-bit value. The first two 16-bit values are returned in `DST1`, and the second two are returned in `DST2`.

Quad 8-Bit Add Subtract

```
add_i4x8(long long src1, char *ptr1, long long src2,
         char *ptr2, int dst1, int dst2);

add_i4x8r(long long src1, char *ptr1, long long src2,
          char *ptr2, int dst1, int dst2);

sub_i4x8(long long src1, char *ptr1, long long src2,
         char *ptr2, int dst1, int dst2);

sub_i4x8r(long long src1, char *ptr1, long long src2,
          char *ptr2, int dst1, int dst2);
```

These macros provide the operations to select two four-byte sequences from the two eight-byte operands provided, add or subtract the corresponding bytes, and generate four 16-bit results. The first two results are stored in `DST1`, and the second two are stored in `DST2`. `PTR1` selects the bytes from `SRC1`, and `PTR2` selects the bytes from `SRC2`. The `add_i4x8r()` and `sub_i4x8r()` variants produce the same instructions as `add_i4x8()` and `sub_i4x8()`, but with the "reverse" option enabled; this swaps the order of the two 32-bit elements in the `SRC` parameters.

Dual 16-Bit Add/Clip

```
int addclip_lo(long long src1, char *ptr1, long long src2, char *ptr2);

int addclip_hi(long long src1, char *ptr1, long long src2, char *ptr2);

int addclip_lor(long long src1, char *ptr1, long long src2, char *ptr2);

int addclip_hir(long long src1, char *ptr1, long long src2, char *ptr2);
```

These operations select two 16-bit values from `src1` using `ptr1`, and two 8-bit values from `src2` using `ptr2`. The pairs are added and then clipped to the range 0 to 255, producing two 8-bit results. The `_lo` versions select bytes 3 and 1 from `src2`, while the `_hi` versions select bytes 2 and 0. The `_lor` and `_hir` versions reverse the order of the 32-bit elements in `src1` and `src2`.

Quad 8-Bit Average

```
int avg_i4x8(long long src1, char *ptr1, long long src2, char *ptr2);

int avg_i4x8_t(long long src1, char *ptr1, long long src2, char *ptr2);

int avg_i4x8_r(long long src1, char *ptr1, long long src2, char *ptr2);

int avg_i4x8_tr(long long src1, char *ptr1, long long src2, char *ptr2);
```

These operations select two 4-byte sequences from `src1` and `src2`, using `ptr1` and `ptr2`. They add the corresponding bytes from each sequence, and then shift each result right once to produce four byte-size averages. There are four variants of the operation to select the reverse and truncate options for the operation.

```
int avg_i2x8_lo  (long long src1, char *ptr1, long long src2);

int avg_i2x8_lot (long long src1, char *ptr1, long long src2);

int avg_i2x8_lor (long long src1, char *ptr1, long long src2);

int avg_i2x8_lotr(long long src1, char *ptr1, long long src2);

int avg_i2x8_hi  (long long src1, char *ptr1, long long src2);

int avg_i2x8_hit (long long src1, char *ptr1, long long src2);

int avg_i2x8_hir (long long src1, char *ptr1, long long src2);

int avg_i2x8_hitr(long long src1, char *ptr1, long long src2);
```

These operations produce two 8-bit average values. Each selects two four-byte sequences from `src1` and `src2` using `ptr`, and then produces averages of the 4-byte sequences as two 2x2-byte clusters. The two results are byte-sized, and are stored in two bytes of the output result; the other two bytes are set to zero. The variants allow for the generation of different options: truncate or round, reverse input pairs, or store results in the low or high bytes of each 16-bit half of the result register.

Accumulator Extract With Addition

```
extract_and_add(long long src1, long long src2, int dst1, int dst2);
```

This macro provides the operation to add the high and low halves of `SRC1` with the high and low halves of `SRC2` to produce two 32-bit results.

Subtract Absolute Accumulate

```
saa(long long src1, char *ptr1, long long src2, char *ptr2,
    int sum1, int sum2, int dst1, int dst2);

saar(long long src1, char *ptr1, long long src2, char *ptr2,
     int sum1, int sum2, int dst1, int dst2);
```

These macros provide the operations to select two 4-byte sequences from `SRC1` and `SRC2`, using `PTR1` and `PTR2` to select. The bytes from `SRC2` are subtracted from their corresponding bytes in `SRC1`, and then the absolute value

of each subtraction is computed. These four results are then added to the four 16-bit values in `SUM1` and `SUM2`, and the results are stored in `DST1` and `DST2` as four 16-bit values.

Example of Use: Sum of Absolute Difference

As an example use of the video operation built-in functions, a block-based video motion estimation algorithm might use sum of absolute difference (SAD) calculations to measure distortion. A reference SAD function may be implemented as:

```
int ref_SAD16x16(unsigned char *image, unsigned char *block, int imgWidth)
{
   int dist = 0;
   int x, y;

   for (y = 0; y < 16; y++) {
      for (x = 0; x < 16; x++)
      dist += abs(image[x] - block[x]);
      image += 16+ (imgWidth-16);
      block += 16;
   }
   return dist;
}
```

Using video operation built-in functions, the code could be written as follows.

> **NOTE:** `imgWidth` should be divisible by 4.

```
int vid_SAD16x16(unsigned char *image, unsigned char *block, int imgWidth)
{
   int x, y;
   long long srcI, srcB;
   int bytesI1, bytesI2, bytesB1, bytesB2;
   int sum1, sum2, res1, res2;
   sum1 = sum2 = 0;
   bytesI2 = bytesB2 = 0;

   /* get 4-byte aligned pointers */
   int *iPtr = ((int)image)&~3;
   int *bPtr = ((int)block)&~3;

   for (y = 0; y < 16; y++) {
      bytesI1 = *iPtr;
      bytesB1 = *bPtr;

      for (x = 0; x < 16; x += 8) {
         iPtr++; bytesI2 = *iPtr++;
         bPtr++; bytesB2 = *bPtr++;

         srcI = compose_i64(bytesI1, bytesI2);
         srcB = compose_i64(bytesB1, bytesB2);
```

```
        saa(srcI, image, srcB, block, sum1, sum2, sum1, sum2);
        bytesI1 = *iPtr;
        bytesB1 = *bPtr;

        srcI = compose_i64(bytesI1, bytesI2);
        srcB = compose_i64(bytesB1, bytesB2);

        saar(srcI, image, srcB, block, sum1, sum2, sum1, sum2);
    }
    iPtr += (imgWidth - 16)/4;
  }
  extract_and_add(sum1, sum2, res1, res2);
  return res1 + res2;
}
```

Misaligned Data Built-In Functions

The following intrinsic functions allow you to explicitly perform loads from misaligned memory locations and stores to misaligned memory locations. On processors with Blackfin cores, these functions generate expanded code to read and write from such memory locations, regardless of whether the access is aligned or not. On processors with Blackfin+ cores, which have hardware support for misaligned accesses, these functions generate normal memory access instructions.

```
#include  <ccblkfn.h>

short misaligned_load16(void *);
short misaligned_load16_vol(volatile void *);
void misaligned_store16(void *, short);
void misaligned_store16_vol(volatile void *, short);

int misaligned_load32(void *);
int misaligned_load32_vol(volatile void *);
void misaligned_store32(void *, int);
void misaligned_store32_vol(volatile void *, int);

long long misaligned_load64(void *);
long long misaligned_load64_vol(volatile void *);
void misaligned_store64(void *, long long);
void misaligned_store64_vol(volatile void *, long long);
```

Note that there are also volatile variants of these functions. Because of the operations required to read from and write to such misaligned memory locations, no assumptions should be made regarding the atomicity of these operations. Refer to #pragma pack (*alignopt)* for more information.

Memory-Mapped Register Access Built-In Functions

The following built-in functions can be used to ensure that the compiler applies any necessary silicon anomaly workarounds for memory-mapped register (MMR) accesses. These workarounds may be necessary for any source that uses non-literal address type accesses (particularly when the (-no-assume-vols-are-mmrs) switch is specified as

---

the compiler is not normally able to identify such code as implementing MMR accesses. An example of this is where an access is made via a pointer whose value cannot be determined at compile time.

The prototypes for the following functions that implement this support are defined in the `ccblkfn.h` include file:

```
unsigned short  mmr_read16(volatile void *); // Performs 16-bit MMR load
unsigned int mmr_read32(volatile void *);    // Performs 32-bit MMR load
void mmr_write16(volatile void *,
                 unsigned short);            // Performs 16-bit MMR store
void mmr_write32(volatile void *,
                 unsigned int);              // Performs 32-bit MMR store
```

The compiler generates equivalent code for uses of these built-in functions as it would for a normal reference of the specified pointer. The only difference when the built-ins are used is that the compiler can ensure that the generated code avoids any silicon anomalies that impact MMR accesses, provided the workarounds are enabled by building for the appropriate silicon revision, or are explicitly enabled via the -workaround *workaround_id[, workaround_id]* switch.

Pragmas

The Blackfin C/C++ compiler supports pragmas. Pragmas are implementation-specific directives that modify the compiler's behavior. There are two types of pragma usage: *pragma directives* and *pragma operators*.

Pragma directives have the following syntax:

#pragma *pragma-directive pragma-directive-operands new-line*

Pragma operators have the following syntax:

_Pragma (*string-literal*)

When processing a pragma operator, the compiler effectively turns it into a pragma directive using a non-string version of *string-literal*. This means that the following pragma directive

#pragma *linkage_name mylinkname*

can also be equivalently expressed using the following pragma operator.

_Pragma ("linkage_name mylinkagename")

The examples in this manual use the directive form.

The compiler issues a warning when it encounters an unrecognized pragma directive or pragma operator.

The following sections describe the supported pragmas.

- Pragmas With Declaration Lists

- Data Declaration Pragmas

- Interrupt Handler Pragmas

- Loop Optimization Pragmas

- General Optimization Pragmas

- Fixed-Point Arithmetic Pragmas

- Inline Control Pragmas

- Linking Control Pragmas

- Function Side-Effect Pragmas

- Format Argument Pragmas

- Class Conversion Optimization Pragmas

- Template Instantiation Pragmas

- Header File Control Pragmas

- Diagnostic Control Pragmas

- Run-Time Checking Pragmas

- Memory Bank Pragmas

- Exceptions Tables Pragma

Pragmas With Declaration Lists

When using pragmas that can be applied to declarations, in most cases, they only affect the immediately-following definition, even if it is part of a list; For example,

```
#pragma align 8
int i1, i2, i3;
```

In the above example, the pragma applies only to `i1`, meaning `i1` is 8-byte aligned, while `i2` and `i3` use the default alignment. The single exception to this is the `section` pragma, which applies to the entire declaration list that follows it; For example,

```
#pragma section("foo")
int x, y, z;
```

In the above example, `x`, `y`, and `z` are placed in section `foo`, and the compiler issues warning cc1738 to allow you to decide whether this is what was intended.

Data Declaration Pragmas

Data declaration pragmas affect the declaration of data and data types.

Data alignment pragmas are used to modify how the compiler arranges data within the processor's memory. Since the Blackfin processor architecture requires memory accesses to be naturally aligned, each data item is normally aligned at least as strongly as itself; two-byte `shorts` have an alignment of 2, and four-byte `longs` have an alignment of 4. An 8-byte `long long` also has an alignment of 4. The Blackfin+ processor architecture contains support for misaligned data accesses, though such misaligned accesses incur a small performance penalty; the compiler maintains normal alignment on Blackfin+ architectures to maximise performance.

When a `struct` is defined, the struct's overall alignment is the same as the field which has the largest alignment. The struct's size may need padding to ensure that all fields are properly aligned and that the struct's overall size is a multiple of its alignment.

Sometimes, it is useful to change these alignments. A `struct` may have its alignment increased to improve the compiler's opportunities in vectorizing access to the data. A `struct` may have its alignment reduced so that a large array occupies less space.

> **NOTE:** On the Blackfin architecture, if a data item's alignment is reduced, the compiler cannot safely access the data item without the risk of causing misaligned memory access exceptions. Programs that use reduced-alignment data must ensure that accesses to the data are made using data types that match the reduced alignment, rather than the default one. For example, if an int has its alignment reduced from the default (4) to 2, it must be accessed as two shorts or four bytes, rather than as a single int
>
> Since the Blackfin+ architecture has support for misaligned data accesses, such partial loads are not required on Blackfin+ processors.

Data alignment pragmas include the `align`, `pack`, and `pad` pragmas. Alignments specified using these pragmas must be a power of two. The compiler rejects uses of those pragmas that specify alignments that are not powers of two.

#pragma align *num*

The `align` pragma can be used before variable declarations and field declarations. It applies to the variable or field declaration that immediately follows the pragma.

The pragma's effect is that the next variable or field declaration is forced to be aligned on a boundary specified by *num*, as follows:

* If the pragma is being applied to a local variable (which will be stored on the stack), the alignment of the variable is changed only when *num* is not greater than the stack alignment, that is 4 bytes. If *num* is greater than the stack alignment, a warning is given that the pragma is being ignored.

* If *num* is greater than the alignment normally required by the following variable or field declaration, the variable or field declaration's alignment is changed to *num*.

* If *num* is less than the alignment normally required, the variable or field declaration's alignment is changed to *num*, and a warning is given that the alignment has been reduced.

The pragma also allows the following keywords as allowable alignment specifications:

`_WORD` specifies a 32-bit alignment

`_LONG` specifies a 64-bit alignment

`_QUAD` specifies a 128-bit alignment

If the `pack` pragma (#pragma pack (*alignopt*)) or `pad` pragma (#pragma pad (*alignopt*)) are currently active, then `align` overrides the immediately-following field declaration.

The following examples show how to use `#pragma align`.

```
struct s{
   #pragma align 8     /* field a aligned on 8-byte boundary     */
   int a;
   int bar;

   #pragma align 16    /* field b aligned on 16-byte boundary    */
   int b;
} t[2];

#pragma align 256
int arr[128];        /* declares an int array with 256 alignment */
```

The following example shows a use that is valid, but emits a compiler warning.

```
#pragma align 1
int warns;          /* declares an int with byte alignment,   */
                    /* causes a compiler warning              */
```

The following is an example of an invalid use of `#pragma align`. Since the alignment is not a power of two, the compiler rejects it and issues an error.

```
#pragma align 3
int errs;       /* INVALID: declares an int with non-power of */
                /* two alignment, causes a compiler error     */
```

> **NOTE:** The align pragma only applies to the immediately-following definition, even if that definition is part of a list.
>
> For example,
> ```
> #pragma align 8
> int i1, i2, i3;        // pragma only applies to i1
> ```

#pragma alignment_region (`alignopt`)

Sometimes it is desirable to specify an alignment for a group of consecutive data items rather than individually. This can be done using the `alignment_region` and `alignment_region_end` pragmas:

- `#pragma alignment_region` sets the alignment for all following data symbols up to the corresponding `alignment_region_end` pragma

- `#pragma alignment_region_end` removes the effect of the active alignment region and restores the default alignment rules for data symbols

The rules concerning the argument are the same as for the `align` pragma (#pragma align *num*). The compiler faults an invalid alignment (such as an alignment that is not a power of two). The compiler warns if the alignment of a data symbol within the control of an `alignment_region` is reduced below its natural alignment (as for `#pragma align`).

Use of the `align` pragma overrides the region alignment specified by the currently active `alignment_region` pragma (if there is one). The currently active `alignment_region` does not affect the alignment of fields.

Example:

```
#pragma align 16
int aa;          /* alignment 16 */
int bb;          /* alignment 4  */

#pragma alignment_region (8)
int cc;          /* alignment 8 */
int dd;          /* alignment 8 */
int ee;          /* alignment 8 */

#pragma align 16
int ff;          /* alignment 16 */
int gg;          /* alignment 8  */
int hh;          /* alignment 8  */

#pragma alignment_region_end
int ii;          /* alignment 4 */

#pragma alignment_region (2)
long double jj;   /* alignment 2, but the compiler warns about the reduction */

#pragma alignment_region_end

#pragma alignment_region (5)
long double kk;   /* the compiler faults this, alignment is not a power of two */

#pragma alignment_region_end
```

#pragma pack (*alignopt*)

The pack pragma may be applied to struct definitions. It applies to all struct definitions that follow, until the default alignment is restored by omitting *alignopt* (for example, by #pragma pack() with empty parentheses).

The pack pragma is used to reduce the default alignment of the struct to be *alignopt*. If fields within the struct have a default alignment greater than align, their alignment is reduced to *alignopt*. If fields within the struct have alignment less than align, their alignment is unchanged.

If *alignopt* is specified, it is illegal to invoke #pragma pad until the default alignment is restored. The compiler generates an error message if the pad and pack pragmas are used in a manner that conflicts.

The following example shows how to use #pragma pack:

```
#pragma pack(1)
/* struct minimum alignment now 1 byte, uses of
   "#pragma pad" would cause a compilation error now */

struct is_packed {
   char a;
   /* normally the compiler would add three padding bytes here,
```

```
      but not now because of prior pragma pack use */
   int b;
} t[2];               /* t definition requires 10 packed bytes */

#pragma pack()
/* struct minimum alignment now, not one byte,
   "#pragma pad"can now be used legally */

struct is_packed u[2];
/* u definition requires 10 packed bytes */
/* struct not_packed is a new type, and will not be packed. */

struct not_packed {
   char a;
   /* compiler will insert three padding bytes here */
   int b;
} w[2];                    /* w definition required 16 bytes */
```

Unlike Blackfin+ processors, the Blackfin processor does not support misaligned memory accesses at the hardware level; the compiler generates additional code to correctly handle reads from (and writes to) misaligned structure members. The code generated will not necessarily be as efficient as reading from (or writing to) an aligned structure member, but that is the trade-off that must be accepted in return for getting packed structures.

Only direct reads from (and writes to) misaligned structure members are automatically handled by the compiler. As a result, taking the address of a misaligned field and assigning it to a pointer causes the compiler to emit a warning. The reason for the warning is that the compiler does not detect a misaligned memory access if the address of a misaligned field is taken and stored in a pointer of a different type to that of the structure.

Since #pragma pack reduces alignment constraints, and therefore reduces the need for padding within the struct, the overall size of the struct can be reduced; in fact, this reduction in size is often the reason for using the pragma. Be aware, however, that the reduced alignment also applies to the struct as a whole, so instances of the struct may start on *alignopt* boundaries instead of the default boundaries of the equivalent unpacked struct.

#pragma pad (*alignopt*)

Th pad pragma may be applied to struct definitions. It applies to struct definitions that follow until the default alignment is restored by omitting *alignopt* (for example, by #pragma pad() with empty parentheses).

The pad pragma is effectively shorthand for placing #pragma align before every field within the struct definition. Like the pack pragma, it reduces the alignment of fields that default to an alignment greater than *alignopt*.

However, unlike the pack pragma, it also increases the alignment of fields that default to an alignment less than *alignopt*.

If *alignopt* is specified, it is illegal to invoke #pragma pack until the default alignment is restored.

The following example shows how to use #pragma pad().

```
#pragma pad(4)
struct {
    int i;
    int j;
} s = {1,2};
#pragma pad()
```

#pragma no_partial_initialization

The `no_partial_initialization` pragma indicates that the compiler should raise a diagnostic if the following structure declaration does not provide an initialization value for all members of the structure. The pragma is useful when a structure declaration is extended between revisions of the software.

The following example shows how to use `#pragma no_partial_initialization`:

```
struct no_err {
    int x;
    int y;
};

#pragma no_partial_initialization
struct with_err {
    int x;
    int y;
};
struct no_err s1 = { 5 };    // no diagnostic
struct with_err s2 = { 5 }; // diagnostic reported
```

Interrupt Handler Pragmas

The `interrupt`, `nmi`, and `exception` pragmas declare that the following function declaration or definition is to be used as an entry in the event vector table (EVT). The compiler arranges for the function to save its context. This is more than the usual called-preserved set of registers. The function returns using an instruction appropriate to the type of event specified by the pragma.

Normally, these pragmas are not used directly; the supported interrupt model uses a dispatcher. Functions that are to be invoked by the interrupt dispatcher should be declared with the `interrupt_dispatched_handler` pragma, which, like the pragmas above, ensures that all used registers including scratch registers are restored, but which does not imply the use of a special return instruction. See the *System Run-Time Documentation* for more information.

Interrupt handler pragmas may be specified on a function's declaration or its definition. Only one of the three pragmas listed above may be specified for a particular function.

The `interrupt_reentrant` pragma is used with the `interrupt` pragma to specify that the function's context-saving prologue should also arrange for interrupts to be re-enabled for the duration of the function's execution.

The `interrupt_level_interrupt` pragmas are also used to specify that a function should be compiled as an interrupt service routine (ISR). Use these pragmas instead of the `interrupt` pragma when compiling interrupt handler functions with the `-isr-imask-check` workaround enabled, or when the workaround is enabled by

default for the targeted processor and silicon revision. These pragmas are supported for interrupt levels 5 (`#pragma interrupt_level_5`) to 15 (`#pragma interrupt_level_15`).

If the `isr-imask-check` workaround is enabled, ISRs declared without explicit interrupt levels-such as those declared using `EX_INTERRUPT_HANDLER()`-check for interrupts occurring while a `CLI` instruction is committed and return immediately if this is detected. They do not attempt to re-raise the interrupt.

Loop Optimization Pragmas

Loop optimization pragmas give the compiler additional information about usage within a particular loop, allowing the compiler to perform more aggressive optimization. These pragmas are placed before the loop statement, and apply to the statement that immediately follows, which must be a `for`, `while`, or `do` statement to have effect. In general, it is most effective to apply loop optimization pragmas to inner-most loops, since the compiler can achieve the most savings there.

The optimizer always attempts to vectorize loops when it is safe to do so. The optimizer exploits the information generated by the interprocedural analysis to increase the cases where it knows it is safe to do so. See Interprocedural Analysis.

Consider the code:

```
void copy(short *a, short *b) {
   int i;
   for (i=0; i<100; i++)
     a[i] = b[i];
}
```

If you call `copy` with two calls, such as `copy(x,y)` and later `copy(y,z)`, interprocedural analysis is unable to tell that "a" never aliases "b". Therefore, the optimizer cannot be sure that one iteration of the loop is not dependent on the data calculated by the previous iteration of the loop. If it is known that each iteration of the loop is not dependent on the previous iteration, then the `vector_for pragma` can be used to explicitly notify the compiler that this is the case.

#pragma all_aligned

The `all_aligned` pragma applies to the subsequent loop. This pragma asserts that all pointers are initially aligned on the most desirable boundary. This pragma has no effect on processors with Blackfin+ architecture.

#pragma different_banks

The `different_banks` pragma allows the compiler to assume that groups of memory accesses based on different pointers within a loop reside in different memory banks. By scheduling them together, memory access performance may be improved.

#pragma loop_count(*min, max, modulo*)

The `loop_count` pragma appears just before the loop it describes. It asserts that the loop iterates at least `min` times, no more than `max` times, and a multiple of `modulo` times. This information enables the optimizer to omit loop guards and to decide whether the loop is worth completely unrolling and whether code needs to be generated for odd iterations. Any of the parameters of the pragma that are unknown may be left blank. For example,

---

```
int i;
#pragma loop_count(24, 48, 8)
for (i=0; i < n; i++)
```

#pragma loop_unroll *N*

The `loop_unroll` pragma can be used only before a `for`, `while`, or `do.. while` loop. The pragma takes one positive integer argument, `N`, and instructs the compiler to unroll the loop `N` times prior to further transforming the code.

In the most general case, the effect of:

```
#pragma loop_unroll N
for ( init statements; condition; increment code )
{
   ... loop_body
}
```

is equivalent to transforming the loop to:

```
for ( init statements; condition;  increment code )
{
               ... loop_body
   /* copy 1 */
   ... increment_code
   if (!condition)
       break;
   ... loop_body
   /* copy 2 */
   ... increment_code
   if (!condition)
       break;
    ... loop_body
   /* copy N-1 */
    ... increment_code
   if (! condition)
       break;

    ... loop_body
    /* copy N */
}
```

Similarly, the effect of:

```
#pragma loop_unroll  N
while ( condition  ) {
   loop_body
}
```

is equivalent to transforming the loop to:

```
while ( condition ) {
    ... loop_body
```

```
    /* copy 1 */
    if (!condition)
        break;
    ... loop_body
    /* copy 2 */
    if (!condition)
         break;
    ... loop_body
    /* copy N-1 */
    if (!condition)
          break;
    ... loop_body
    /* copy N */
}
```

and the effect of:

```
#pragma loop_unroll N
do {
      loop_body
} while ( condition )
```

is equivalent to transforming the loop to:

```
do {
   loop_body
   /* copy 1 */
   if (!condition)
        break;

    ... loop_body
   /* copy 2 */
   if (! condition)
        break;
    ... loop_body
   /* copy N-1 */
   if (!condition)
        break;
    ... loop_body
   /* copy N */
} while ( condition   )
```

#pragma no_vectorization

When specified on a loop, the `no_vectorization` pragma turns off all vectorization for the loop.

This pragma may also be specified on a function definition. For more information, see #pragma no_vectorization.

#pragma no_alias

Use the `no_alias` pragma to inform the compiler that the following loop has no loads or stores that conflict. When the compiler finds memory accesses that potentially refer to the same location through different pointers

(known as *aliases*) the compiler is restricted in how it can reorder or vectorize the loop, because all the accesses from earlier iterations must be complete before the compiler can arrange for the next iteration to start.

For example,

```
void vadd(int *a, int *b, int *out, int n) {
  int i;
  #pragma no_alias
  for (i=0; i < n; i++)
    out[i] = a[i] + b[i];
}
```

The `no_alias` pragma appears just before the loop it describes. This pragma asserts that in the next loop, no load or store operations conflict with each other. In other words, no load or store in any iteration of the loop has the same address as any other load or store in the current or in any other iteration of the loop. In the example above, if pointers `a` and `b` point to two memory areas that do not overlap, no load from `b` is using the same address as any store to `a`. Therefore, `a` is never an alias for `b`.

Using the `no_alias` pragma can lead to better code because it allows any number of iterations to be performed concurrently (rather than just two at a time), thus providing better software pipelining by the optimizer.

#pragma vector_for

The `vector_for` pragma notifies the optimizer that it is safe to execute two iterations of the loop in parallel. The `vector_for` pragma does not force the compiler to vectorize the loop. The optimizer checks various properties of the loop and does not vectorize it if it believes to be unsafe or if it cannot deduce that the various properties necessary for the vectorization transformation are valid.

Strictly speaking, the pragma simply disables checking for loop-carried dependencies.

```
void copy(short *a, short *b) {
   int i;
   #pragma vector_for
   for (i=0; i100; i++)
      a[i] = b[i];
}
```

In cases where vectorization is impossible (for example, if array `a` is aligned on a word boundary but array `b` is not, and the target processor uses the Blackfin architecture rather than the Blackfin+ architecture), the information given in the assertion made by `vector_for` may still be put to good use in aiding other optimizations.

General Optimization Pragmas

The compiler supports several pragmas which can change the optimization level while a given module is being compiled. These pragmas must be used globally, immediately prior to a function definition. The pragmas do not just apply to the immediately-following function; they remain in effect until the end of the compilation, or until they are superseded by one of the following `optimize_` pragmas.

- `#pragma optimize_off` - This pragma turns off the optimizer, if it was enabled. It has the same effect as compiling with no optimization enabled.

- `#pragma optimize_for_space` - This pragma turns on the optimizer, if it was disabled, or sets the focus to give reduced code size a higher priority than high performance, where these conflict.

- `#pragma optimize_for_speed` - This pragma turns on the optimizer, if it was disabled, or sets the focus to give high performance a higher priority than reduced code size, where these conflict.

- `#pragma optimize_as_cmd_line` - This pragma resets the optimization settings to be those specified on the `ccblkfn` command line when the compiler was invoked.

The following are code examples of `optimize_` pragmas.

```
#pragma optimize_off
void non_op() { /* non-optimized code */ }

#pragma optimize_for_space
void op_for_si() { /* code optimized for size */ }

#pragma optimize_for_speed
void op_for_sp() { /* code optimized for speed */ }
/* subsequent functions declarations optimized for speed */
```

Fixed-Point Arithmetic Pragmas

The compiler supports several pragmas to change the semantics of arithmetic on the native fixed-point types `fract` and `accum`. These are `#pragma FX_CONTRACT {ON|OFF}` and `#pragma FX_ROUNDING_MODE {TRUNCATION|BIASED|UNBIASED}`. In addition, `#pragma STDC FX_FULL_PRECISION {ON|OFF|DEFAULT}`, `#pragma STDC FX_FRACT_OVERFLOW {SAT|DEFAULT}`, and `#pragma STDC FX_ACCUM_OVERFLOW {SAT|DEFAULT}` are accepted by the compiler but have no effect on generated code.

These pragmas may be used at file scope, in which case they apply to all following functions until another pragma is respecified to change the pragma state. Alternatively, they may be specified in a `{ }` delimited scope (or compound statement), where they will temporarily override the current setting of the pragma's state until the end of the scope.

For more information, see Using Native Fixed-Point Types.

#pragma FX_CONTRACT {ON|OFF}

The `FX_CONTRACT {ON|OFF}` pragma may be used to control the precision of intermediate results of calculations on the native fixed-point types `fract` and `accum`. If `FX_CONTRACT` is `ON`, where an intermediate result is not stored back to a named variable, the compiler may choose to keep the intermediate result in greater precision than that mandated by the ISO/IEC C Technical Report 18037. It will do this where maintaining the higher precision allows more efficient code to be generated.

When `FX_CONTRACT` is `OFF`, the compiler adheres strictly to the ISO/IEC Technical Report 18037 and converts all intermediate results to the type dictated in this standard before use.

The following example shows the use of this pragma.

```
accum mac(accum a, fract f1, fract f2) {
   #pragma FX_CONTRACT ON
   a += f1 * f2;   /* compiler creates multiply-accumulate instruction */
```

```
    return a;
}
```

The default state of the `FX_CONTRACT` pragma is `ON`.

#pragma FX_ROUNDING_MODE {TRUNCATION|BIASED|UNBIASED}

The `FX_ROUNDING_MODE {TRUNCATION|BIASED|UNBIASED}` pragma may be used to control the rounding mode used during calculations on the native fixed-point types `fract` and `accum`.

When `FX_ROUNDING_MODE` is set to `TRUNCATION`, the exact mathematical result of a computation is rounded by truncating the least significant bits beyond the precision of the result type. This is equivalent to rounding towards negative infinity.

When `FX_ROUNDING_MODE` is set to `BIASED`, the exact mathematical result of a computation is rounded to the nearest value that fits in the result type. If the exact result lies exactly half-way between two consecutive values in the result type, the result is rounded up to the higher one. Note that this rounding mode pragma should be used in conjunction with the `set_rnd_mod_biased()` built-in function. For more information, see Changing the RND_MOD Bit.

When `FX_ROUNDING_MODE` is set to `UNBIASED`, the exact mathematical result of a computation is rounded to the nearest value that fits in the result type. If the exact result lies exactly half-way between two consecutive values in the result type, the result is rounded to the even value. Note that this rounding mode pragma should be used in conjunction with the `set_rnd_mod_unbiased()` built-in function. For more information, see Changing the RND_MOD Bit.

The following example shows the use of this pragma.
```
fract divide_biased(fract f1, fract f2) {
   #pragma FX_ROUNDING_MODE BIASED
   set_rnd_mod_biased();
   return f1 / f2;   /* compiler creates divide with biased rounding */
}
```

The default state of the `FX_ROUNDING_MODE` pragma is `TRUNCATION`.

#pragma STDC FX_FULL_PRECISION {ON|OFF|DEFAULT}

The `STDC FX_FULL_PRECISION {ON|OFF|DEFAULT}` pragma is used by the ISO/IEC Technical Report 18037 to permit an implementation to generate faster code for fixed-point arithmetic, but produce lower-accuracy results.

The compiler always produces full-accuracy results. Therefore, although the pragma is accepted by the compiler, the code generated will be the same regardless of the state of `FX_FULL_PRECISION`.

#pragma STDC FX_FRACT_OVERFLOW {SAT|DEFAULT}

The `STDC FX_FRACT_OVERFLOW {SAT|DEFAULT}` pragma is used by the ISO/IEC Technical Report 18037 to permit an implementation to generate code that does not saturate `fract`-typed results on overflow.

`fract` arithmetic with the CCES compiler always saturates on overflow. Therefore, although the pragma is accepted by the compiler, the code generated will be the same regardless of the state of `FX_FRACT_OVERFLOW`.

#pragma STDC FX_ACCUM_OVERFLOW {SAT|DEFAULT}

The `STDC FX_ACCUM_OVERFLOW {SAT|DEFAULT}` pragma is used by the ISO/IEC Technical Report 18037 to permit an implementation to generate code that does not saturate `accum`-typed results on overflow.

`accum` arithmetic with the CCES compiler always saturates on overflow. Therefore, although the pragma is accepted by the compiler, the code generated will be the same regardless of the state of `FX_ACCUM_OVERFLOW`.

Inline Control Pragmas

The compiler supports three pragmas to control the inlining of code (#pragma always_inline, #pragma inline, and #pragma never_inline). Additionally, #pragma source_position_from_call_site affects the debug information on inlined functions.

#pragma always_inline

The `always_inline` pragma may be applied to a function definition to indicate to the compiler that the function should always be inlined, and never called "out of line". The pragma may only be applied to function definitions with the `inline` qualifier, and may not be used on functions with variable-length argument lists. This pragma is not valid for function definitions that have interrupt-related pragmas associated with them.

If the function in question has its address taken, the compiler cannot guarantee that all calls are inlined, so a warning is issued.

See Function Inlining for details of pragma precedence during inlining

The following are examples of the `always_inline` pragma.

```
int func1(int a) {          // only consider inlining
   return a + 1;            // if -Oa switch is on
}
inline int func2(int b) {   // probably inlined, if optimizing
   return b + 2;
}

#pragma always_inline
inline int func3(int c) {   // always inline, even unoptimized
   return c + 3;
}

#pragma always_inline
int func4(int d) {          // error: not an inline function
   return d + 4;
}
```

#pragma inline

The `inline` pragma instructs the compiler to inline the function if it is considered desirable. The pragma is equivalent to specifying the `inline` keyword, but may be applied when the `inline` keyword is not allowed (such as when compiling in MISRA-C mode). For more information, see MISRA-C Compiler.

```
#pragma inline
int func5(int a, int b) {   /* can be inlined */
```

```
    return a / b;
}
```

#pragma never_inline

The `never_inline` pragma can be applied to a function definition to indicate to the compiler that the function should always be called "out of line", and that the function's body should never be inlined.

This pragma may not be used on function definitions that have the `inline` qualifier.

See Function Inlining for details of pragma precedence during inlining.

The following are code examples for the `never_inline` pragma.

```
#pragma never_inline
int func5(int e) {             // never inlined, even with -Oa switch
    return e + 5;
}

#pragma never_inline
inline int func5(int f) {   // error: inline function
    return f + 6;
}
```

#pragma source_position_from_call_site

This pragma may be applied to function definitions. It affects the debug information emitted when a function is inlined: instead of using the actual source code position of each inlined statement, the position of the call to the inlined function is attached to code generated for the function.

When stepping through a program in the debugger, this can be useful on very small inlined functions, so as to avoid the current position jumping between the code under investigation and the definitions of inlined functions.

The pragma is used in the `builtins.h` header for compiler built-in functions. It does not affect the generated machine code.

Linking Control Pragmas

Linking control pragmas (`linkage_name`, `additional_linkage_name`, `function_name`, `core`, `retain_name`, `section`, `file_attr`, `symbolic_ref`, and `weak_entry`) change how a given global function or variable is viewed during the linking stage.

#pragma linkage_name *identifier*

The `linkage_name` pragma associates the *identifier* with the next external function declaration. It ensures that the *identifier* is used as the external reference, instead of following the compiler's usual conventions. If the *identifier* is not a valid function name, as could be used in normal function definitions, the compiler generates an error. See also the (asm) Keyword for Specifying Names in Generated Assembler keyword.

The following example shows the use of this pragma.

```
#pragma linkage_name realfuncname
void funcname ();
void func() {
```

```
    funcname();    /* compiler will generate a call to realfuncname */
}
```

See also #pragma function_name *identifier* .

#pragma function_name *identifier*

This pragma associates the identifier with the next external function declaration. The compiler uses the given identifier instead of the one used in the function declaration for any external references. However, unlike `#pragma linkage_name`, the compiler follows its usual conventions to mangle the new name, for example by prefixing a leading underscore. For example,

```
_Pragma("function_name realfuncname")
void funcname ();
```

See also #pragma linkage_name *identifier* .

#pragma additional_linkage_name *identifier*

The `additional_linkage_name` pragma associates the `identifier` with the next external function definition. It ensures that the `identifier` is used as an additional linkage name for the function, by generating a `.set` assembly directive to create an alias symbol. This allows the same function to be called through multiple names without indirection overhead, and can be useful for example when a function name is changed but backward compatibility needs to be ensured.

The following example shows the use of this pragma.

```
#pragma additional_linkage_name _func_alias
void func() {
    /* ... */
}
```

See also #pragma linkage_name *identifier* .

#pragma core

When building a project that targets multiple processors or multiple cores on a processor, a link stage may produce executables for more than one core or processor. The interprocedural analysis (IPA) framework requires that some conventions be adhered to in order to successfully perform its analysis for such projects.

Because the IPA framework collects information about the whole program, including information on references which may be to definitions outside the current translation unit, the IPA framework must be able to distinguish these definitions and their references without ambiguity.

If any confusion were allowed about which definition a reference refers to, then the IPA framework could potentially cause bad code to be generated, or could cause translation units in the project to be continually recompiled. Global symbols are relevant in this respect. The IPA framework correctly handles locals and static symbols because multiple definitions are not possible within the same file, so there can be no ambiguity.

In order to disambiguate all references and the definitions to which they refer, each definition within a given project must have a unique name. It is illegal to define two different functions or variables with the same name. This is illegal in single-core projects because this would lead to multiple definitions of a symbol and the link would fail. In

---

multi-core projects, however, it maybe possible to link a project with multiple definitions because one definition could be linked into each link project, resulting in a valid link. Without detailed knowledge of what actions the linker had performed, however, the IPA framework would not be able to disambiguate such multiple definitions. For this reason, to use the IPA framework, you must ensure unique names even in projects targeting multiple cores or processors.

There are a few cases for which it is not possible to ensure unique names in multi-core or multi-processor projects. One such case is `main`. Each processor or core will have its own `_main` function, and these need to be disambiguated for the IPA framework to be able to function correctly. Another case is where a library (or the C Run-time start up) references a symbol which the user may wish to define differently for each core.

For this reason, the `#pragma core( corename )` is provided.

The `core` pragma can be provided immediately prior to a definition or a declaration. The pragma allows you to give a unique identifier to each definition. It also allows you to indicate to which definition each reference refers. The IPA framework uses this core identifier to distinguish all instances of symbols with the same name and will therefore be able to carry out its analysis correctly.

> **NOTE:** The specified *corename*, which is case-sensitive, must consist of alphanumeric characters only.

Use the `core` pragma on:

- Every definition (not in a library) for which there needs to be a distinct definition for each core.

- Every declaration of a symbol (not in a library) for which the relevant definition includes the use of `#pragma core`. The core specified for a declaration must agree with the core specified for the definition.

The IPA framework will not need to be informed of any distinction if there are two identical copies of the same function or data with the same name. Functions or data that come from objects and that are duplicated in memory local to each core, for example, will not need to be distinguished. The IPA framework does not need to know exactly which instance each reference will get linked to because the information processed by the framework is identical for each copy. Essentially, the pragma only needs to be specified on items where there will be different functions or data with the same name incorporated into the executable for each core.

The following example of `#pragma core` usage distinguishes two different main functions:

```
/* foo.c */
#pragma core("coreA")
int main(void) {          /* Code to be executed by core A */
}
/* bar.c */
#pragma core("coreB")
int main(void) {          /* Code to be executed by core B */
}
```

Omitting either instance of the pragma will cause the IPA framework to issue a fatal error, indicating that the pragma has been omitted on at least one definition.

The following example issues an error because the name contains a non-alphanumeric character:

```
#pragma core("core/A")
int main(void) {             /* Code to executed on core A */
}
```

In the following example, the `core` pragma must be specified on a declaration as well as the definitions. A library contains a reference to a symbol, which is expected to be defined for each core. Two more modules define the main functions for the two cores. Two further modules, each only used by one of the cores, references this symbol, and therefore require the pragma.

```
/* libc.c */
#include <stdio.h>
extern int core_number;
void print_core_number(void) {
   printf("Core %d\n", core_number);
}
/* maina.c */
extern void fooa(void);
#pragma core("coreA")
int core_number = 1;
#pragma core("coreA")
int main(void) {
   /* Code to be executed by core A */
   print_core_number();
   fooa();
}
/* mainb.c */
extern void foob(void);
#pragma core("coreB")
int core_number = 2;
#pragma core("coreB")
int main(void) {
  /* Code to be executed by core B */
   print_core_number();
   foob();
}
/* fooa.c */
#include <stdio.h>
#pragma core("coreA")
extern int core_number;
void fooa(void) {
   printf("Core: is core%c\n", 'A' - 1 + core_number);
}
/* foob.c */
#include <stdio.h>
#pragma core("coreB")
extern int core_number;
void fooa(void) {
   printf("Core: is core%c\n", 'A' - 1 + core_number);
}
```

In general, it is only necessary to use #pragmacore in this manner when there is a reference from outside the application (in a library, for example) where there is expected to be a distinct definition provided for each core, and where there are other modules that also require access to their respective definition. Notice also that the declaration of core_number in lib.c does not require the use of the `core` pragma because it is part of a translation unit to be included in a library.

A project that includes more than one definition of `main` will undergo extra checking to catch problems that would otherwise occur in the IPA framework. For any non-template symbol that has more than one definition, the tool chain will fault any definitions that are outside libraries that do not specify a core name with the `core` pragma. This check does not affect the normal behavior of the prelinker with respect to templates and in particular the resolution of multiple template instantiations.

To clarify:

Inside a library, `#pragma core` is not required on declarations or definitions of symbols that are defined more than once. However, a library can be responsible for forcing the application to define a symbol more than once (that is, once for each core). In this case, the definitions and declarations require the `core` pragma to be used outside the library to distinguish the multiple instances.

> **NOTE:** The tool chain cannot check that uses of `#pragma core` are consistent. If you use the pragma inconsistently or ambiguously, the IPA framework may cause incorrect code to be generated or may cause continual recompilation of the application's files.

It is also important to note that the `core` pragma does not change the linkage name of the symbol it is applied to in any way.

For more IPA information, see Interprocedural Analysis.

#pragma retain_name

The `retain_name` pragma indicates that the function or variable declaration that follows the pragma is not to be removed even though it has no apparent use. Normally, when interprocedural analysis or linker elimination are enabled, the CCES tools identify unused functions and variables and will eliminate them from the resulting executable to reduce memory requirements. The `retain_name` pragma instructs the tools to retain the specified symbol regardless.

The following example shows how to use this pragma.
```
int delete_me(int x) {
    return x-2;
}
#pragma retain_name
int keep_me(int y) {
    return y+2;
}
int main(void) {
    return 0;
}
```

Since the program has no uses for `delete_me()` or `keep_me()`, the compiler removes `delete_me()`, but keeps `keep_me()` because of the pragma. You do not need to specify `retain_name` for `main()`.

The pragma is only valid for global symbols. It is not valid for the following kinds of symbols:

- Symbols with `static` storage class

- Function parameters

- Symbols with `auto` storage class (locals). These are allocated on the stack at run time.

- Members/fields within `structs/unions/classes`

- Type declarations

For more information on IPA, see Interprocedural Analysis.

#pragma section/#pragma default_section

The `section` pragma and `default_section` pragma provide greater control over the sections in which the compiler places symbols.

The `section(SECTSTRING[, QUALIFIER, ...])` pragma is used to override the target section for any global or static symbol immediately following it. The pragma allows greater control over section qualifiers compared to the `section` keyword.

The `default_section(SECTKIND[, SECTSTRING[, QUALIFIER, ...]])` pragma is used to override the default sections in which the compiler is placing its symbols.

The default sections fall into the categories listed under *SECTKIND*. Except for the `STI` category, this pragma remains in force for a section category until its next use with that particular category, or the end of the file. The `STI` is an exception, in that only one `STI` `default_section` can be specified and its scope is the entire file scope, not just the part following the use of `STI`. A warning is issued if several `STI` sections are specified in the same file.

The omission of a section name results in the default section being reset to be the section that was in use at the start of the file, which can be either a compiler default value, or a value set by the user through the `-section` command-line switch (for example, `-section SECTKIND=SECTSTRING`).

In all cases (including `STI`), the `default_section` pragma overwrites the value specified with the `-section` command line switch.

```
#pragma default_section(DATA, "NEW_DATA1")
int x;
#pragma default_section(DATA, "NEW_DATA2")
int y=5;
#pragma default_section(DATA, "NEW_DATA3")
int z;
```

In this case, `y` is placed in `NEW_DATA2` because the definition of `y` is within its scope.

A `default_section` pragma can only be used at global scope, where global variables are allowed.

SECTKIND can be one of the keywords shown in the **SECTKIND Keywords** table.

Table 2-35: SECTKIND Keywords

Keyword	Description
CODE	Section is used to contain procedures and functions
ALLDATA	Shorthand notation for DATA, CONSTDATA, BSZ, STRINGS, and AUTOINIT
DATA	Section is used to contain "normal data"
CONSTDATA	Section is used to contain read-only data
BSZ	Section is used to contain zero-filled data
SWITCH	Section is used to contain jump tables to implement C/C++ switch statements
VTABLE	Section is used to contain C++ virtual-function tables
STI	Section that contains code required to be executed by C++ initializations. For more information, see Constructors and Destructors of Global Class Instances.
STRINGS	Section that stores string literals
AUTOINIT	Contains data used to initialize aggregate autos

SECTSTRING is a double-quoted string containing the section name, exactly as it will appear in the assembler file.

Changing one section kind has no effect on other section kinds. For instance, even though STRINGS and CONSTDATA are, by default, placed by the compiler in the same section, if the default section for CONSTDATA is changed, the change has no effect on the STRINGS data.

Note that ALLDATA is not a real section, but rather pseudo-kind that stands for DATA, CONSTDATA, STRINGS, AUTOINIT, and BSZ. Changing ALLDATA is equivalent to changing all of these section kinds.

Therefore,

```
#pragma default_section(ALLDATA, params)
```

is equivalent to the sequence:

```
#pragma default_section(DATA, params)
#pragma default_section(CONSTDATA, params)
#pragma default_section(STRINGS, params)
#pragma default_section(AUTOINIT, params)
#pragma default_section(BSZ, params)
```

QUALIFIER can be one of the keywords in the ***QUALIFIER Keywords*** table.

Table 2-36: QUALIFIER Keywords

Keyword	Description
ZERO_INIT	Section is zero-initialized at program startup
NO_INIT	Section is not initialized at program startup
RUNTIME_INIT	Section is user-initialized at program startup

Table 2-36: QUALIFIER Keywords (Continued)

Keyword	Description
DOUBLE32	Section may contain 32-bit but not 64-bit `doubles`
DOUBLE64	Section may contain 64-bit but not 32-bit `doubles`
DOUBLEANY	Section may contain either 32-bit or 64-bit `doubles`

There may be any number of comma-separated section qualifiers within such pragmas, but they must not conflict with one another. Qualifiers must also be consistent across pragmas for identical section names, and omission of qualifiers is not allowed, even if at least one such qualifier has appeared in a previous pragma for the same section. If any qualifiers have not been specified for a particular section by the end of the translation unit, the compiler uses default qualifiers appropriate for the target processor.

The following specifies that `f()` should be placed in a section `foo` which is `DOUBLEANY` qualified:

```
#pragma section("foo", DOUBLEANY)
void f() {}
```

The compiler always tries to honor the `section` pragma as its highest priority, and the `default_section` pragma is always the lowest priority of the two.

For example, the following code results in function `f` being placed in the section `foo`:

```
#pragma default_section(CODE, "bar")
#pragma section("foo")
void f() {}
```

The following code results in `x` being placed in section `zeromem`:

```
#pragma default_section(BSZ, "zeromem")
int x;
```

> **NOTE:** In cases where a C++ STL object is required to be placed in a specific memory section, using #pragma section/default_section does not work. Instead, a non-default heap must be used as explained in Allocating C++ STL Objects to a Non-Default Heap.

#pragma file_attr("name[=value]" [, "name[=value]" [...]])

The `file_attr` pragma directs the compiler to emit the specified attributes when it compiles a file containing the pragma. Multiple `#pragma file_attr` directives are allowed in one file.

If `=value` is omitted, the default value of `1` is used.

> **CAUTION:** The value of an attribute is all the characters after the = symbol and before the closing **"** symbol, including spaces. A warning will be emitted by the compiler if you have a preceding or trailing space as an attribute value, as this is likely to be a mistake.

See File Attributes for more information on using attributes.

#pragma symbolic_ref

The `symbolic_ref` pragma can be used before a public global variable, to indicate to the compiler that references to that variable should only be through the variable's symbolic name. Loading the address of a variable into a pointer register can be an expensive operation, and the compiler usually avoids this when possible. Consider the case where

```
int x;
int y;
int z;
void foo(void) { x = y + z; }
```

Given that the three variables are in the same data section, the compiler can generate the following code:

```
_foo:
    P0.L = .epcbss;
    P0.H = .epcbss;
    R0 = [P0+ 4];
    R1 = [P0+ 8];
    R0 = R1 + R0;
    [P0+ 0] = R0;
    RTS;

    .section/ZERO_INIT bsz;

    .align 4;
    .epcbss:
    .type .epcbss,STT_OBJECT;
    .byte _x[4];
    .global _x;
    .type _x,STT_OBJECT;
    .byte _y[4];
    .global _y;
    .type _y,STT_OBJECT;
    .byte _z[4];
    .global _z;
    .type _z,STT_OBJECT;
.epcbss.end:
```

Having loaded a pointer to x (which shares the address of the start of the `.epcbss` section), the compiler can use offsets from this pointer to access "y" and "z", avoiding the expense of loading addresses for those variables. However, this forces the linker to ensure that the relative offsets between x, y, z, and `.epcbss` do not change during the linking process.

There are cases when you might wish the compiler to reference a variable only through its symbolic name, such as when you are using `RESOLVE()` in the `.ldf` file to explicitly map the variable to a particular address. The compiler automatically uses symbolic references for:

- Volatile variables

- Variables specified with `#pragma weak_entry`

- Variables greater than or equal to 16 bytes in size

If other cases arise, you can use `#pragma symbolic_ref` to explicitly request this behavior. For example,

```
int x;
#pragma symbolic_ref
int y;
int z;
void foo(void) { x = y + z; }
```

produces:

```
_foo:
    P0.L = .epcbss;
    I0.L = _y;
    P0.H = .epcbss;
    I0.H = _y;
    MNOP || R0 = [P0+ 4] || R1 = [I0];
    R0 = R0 + R1;
    [P0+ 0] = R0;
    RTS;

    .section/ZERO_INIT bsz;

    .align 4;
    .epcbss:
    .type .epcbss,STT_OBJECT;
    .byte _x[4];
    .global _x;
    .type _x,STT_OBJECT;
    .byte _z[4];
    .global _z;
    .type _z,STT_OBJECT;
    .epcbss.end:
    .align 4;
    .global _y;
    .type _y,STT_OBJECT;
    .byte _y[4];
._y.end:
```

Note that variable `y` is referenced explicitly by name, rather than using the common pointer to `.epcbss`, and it is declared outside the bounds of the (`.epcbss`, `.epcbss.end`) pair. The (`_y`, `._y.end`) form a separate pair that can be moved by the linker, if necessary, without affecting the functionality of the generated code.

The `symbolic_ref` pragma can only be used immediately before declarations of global variables, and only applies to the immediately-following declaration.

#pragma weak_entry

The `weak_entry` pragma may be used before a static variable or function declaration or definition. It applies to the function/variable declaration or definition that immediately follows the pragma. Use of this pragma causes the compiler to generate the function or variable definition with weak linkage.

The following are example uses of the `#pragma weak_entry` directive.

```
#pragma weak_entry
int w_var = 0;

#pragma weak_entry
void w_func(){}
```

> **NOTE:** When a symbol definition is weak, it may be discarded by the linker in favor of another definition of the same symbol. Therefore, if any modules in the application use the weak_entry pragma, interprocedural analysis is disabled because it would be unsafe for the compiler to predict which definition will be selected by the linker. For more information, see Interprocedural Analysis.

Function Side-Effect Pragmas

Function side-effect pragmas (`alloc`, `pure`, `const`, `inline`, `misra_func`, `no_vectorization`, `noreturn`, `exceptret`, `regs_clobbered`, `regs_clobbered_call`, `overlay`, `pgo_ignore` and `result_alignment`) are used before a function declaration to give the compiler additional information about the function to improve the code surrounding the function call. These pragmas should be placed before a function declaration and should apply to that function. For example,

```
#pragma pure
long dot(short*, short*, int);
```

#pragma alloc

The `alloc` pragma tells the compiler that the function behaves like the library function `malloc`, returning a pointer to a newly allocated object. An important property of these functions is that the pointer returned by the function does not point at any other object in the context of the call.

In the following example, the compiler can reorder the iterations of the loop because the `#pragma alloc` tells it that p and q cannot overlap.

```
#pragma alloc
short *new_buf(void);
short *copy_buf(short *a) {
   int i;
   short * p = a;
   short * q = new_buf();
   for (i=0; i<100; i++)
      *p++ = *q++;
   return p;
}
```

The GNU attribute malloc is also supported with the same meaning.

#pragma const

The `const` pragma is a more restrictive form of the `pure` pragma (#pragma pure). It tells the compiler that the function does not read from global variables, does not write to them, or read or write volatile variables. The result is therefore a function of its parameters. If any parameters are pointers, the function may not read the data they point at.

#pragma exceptret

The exceptret pragma can be placed before a function prototype or definition. The pragma tells the compiler that the function to which it applies will not return normally, but that it might throw a C++ exception.

The use of this pragma allows the compiler to treat any code, other than exception handlers, that follows a call to a function declared with the pragma as unreachable and hence removable.

```
#pragma exceptret
void fail() {
    throw std::runtime_error("Something went wrong.");
}

main() {
    fail();
    /* any code here will be removed */
}
```

#pragma inline

The inline pragma is placed before a function prototype or definition. It tells the compiler that this function is to be treated as inline.

#pragma misra_func(*arg*)

The misra_func pragma is placed before a function prototype. It is used to support MISRA-C rules 20.4, 20.7, 20.8, 20.9, 20.10, 20.11, and 20.12. The *arg* indicates the type of function with respect to the MISRA-C rule. Functions following rule 20.4 would take arg heap, 20.7 arg jmp, 20.8 arg handler, 20.9 arg io, 20.10 arg string_conv, 20.11 arg system, and 20.12 arg time.

#pragma no_vectorization

When specified on a function, the no_vectorization pragma turns off all vectorization for all loops in the function.

This pragma may also be specified on a loop. For more information, see #pragma no_vectorization.

#pragma noreturn

The noreturn pragma can be placed before a function prototype or definition. It tells the compiler that the function to which it applies will never return to its caller. For example, a function such as the standard C function exit never returns. If the function can throw a C++ exception, the #pragma exceptretpragma should be used instead.

The use of this pragma allows the compiler to treat all code following a call to a function declared with the pragma as unreachable and hence removable.

```
#pragma noreturn
void func() {
    while(1);
}

main() {
    func();
```

```
    /* any code here will be removed */
}
```

#pragma pgo_ignore

The `pgo_ignore` pragma tells the compiler that no profile should be generated for this function when using profile-guided optimization. This is useful when the function is concerned with error checking or diagnostics.

For example,

```
extern const short *x, *y;
int dotprod(void) {
   int i, sum = 0;
   for (i = 0; i < 100; i++)
      sum += x[i] * y[i];
   return sum;
}

#pragma pgo_ignore
int check_dotprod(void) {
   /* The compiler will not profile this comparison */
   return dotprod() == 100;
}
```

#pragma pure

The `pure` pragma tells the compiler that the function does not write to any global variables, and does not read or write any volatile variables. Its result, therefore, is a function of its parameters or of global variables. If any of the parameters are pointers, the function may read the data they point at but may not write to the data.

Since the function call has the same effect every time it is called (between assignments to global variables), the compiler need not generate the code for every call.

Therefore, in the following example, the compiler can replace the ten calls to `sdot` with a single call made before the loop.

```
#pragma pure
long sdot(short *, short *, int);

long tendots(short *a, short *b, int n) {
   int i;
   long s = 0;
      for (i = 1; i < 10; ++i)
         s += sdot(a, b, n);  // call can get hoisted out of loop
   return s;
}
```

#pragma regs_clobbered *string*

The `regs_clobbered` pragma may be used with a function declaration or definition to specify which registers are modified (or clobbered) by that function. The *string* contains a list of registers and is case-insensitive.

When used with an external function declaration, this pragma acts as an assertion, telling the compiler something it would not be able to discover for itself.

In the following example, the compiler knows that only registers `r5`, `p5`, and `i3` may be modified by the call to `f`, so it may keep local variables in other registers across that call.

```
#pragma regs_clobbered "r5 p5 i3"
void f(void);
```

The `regs_clobbered` pragma may also be used with a function definition, or a declaration preceding a definition (when it acts as a command to the compiler to generate register saves, and restores on entry and exit from the function) to ensure it only modifies the registers in string.

For example,

```
#pragma regs_clobbered "r3 m4 p5"
int g(int a) {
    return a+3;
}
```

> **NOTE:** The `regs_clobbered` pragma may not be used in conjunction with `#pragma interrupt`. If both pragmas are specified, a warning is issued and the `regs_clobbered` pragma is ignored.

To obtain optimal results with the pragma, it is best to restrict the clobbered set to be a subset of the default scratch registers. When considering when to apply the `regs_clobbered` pragma, it may be useful to look at the output of the compiler to see how many scratch registers were used. Restricting the volatile set to these registers will produce no impact on the code produced for the function but may free up registers for the caller to allocate across the call site.

> **NOTE:** The `regs_clobbered` pragma cannot be used in any way with pointers to functions. A function pointer cannot be declared to have a customized clobber set, and it cannot take the address of a function which has a customized clobber set. The compiler raises an error if either of these actions are attempted.

String Syntax

A `regs_clobbered` *string* consists of a list of registers, register ranges, or register sets that are clobbered. Items in the list are separated by spaces, commas, or semicolons.

A *register* is a single register name-the same name may be used in an assembly file.

A *register range* consists of `start` and `end` registers, which reside in the same register class, separated by a hyphen. All registers between the two (inclusive) are clobbered.

A *register set* is a name for a specific set of commonly-clobbered registers that is predefined by the compiler. The ***Clobbered Register Sets*** table shows defined clobbered register sets.

Table 2-37: Clobbered Register Sets

Set	Registers
Pscratch	General addressing registers that are scratch by default
DAGscratch	DAG addressing registers that are scratch by default
CCset	ASTAT register
Dscratch	General data registers that are scratch by default
DPscratch	All addressing registers that are scratch by default
ALLscratch	Entire default scratch register set
everything	All registers, apart from those that are user-reserved or unclobberable

When the compiler detects an illegal string, a warning is issued and the default volatile set is used instead. (See Scratch Registers.)

Unclobberable and Must-Clobber Registers

There are certain caveats as to what registers may or must be placed in the clobbered set.

On Blackfin processors, the SP and FP registers may not be specified in the clobbered set, as the correct operation of the function call requires their values to be preserved. If the user specifies them in the clobbered set, a warning is issued and they are removed from the specified clobbered set.

Registers from the following classes may be specified in the clobbered set, and code is generated to save them as necessary.

```
I, P, D, M, ASTAT, A0, A1, LC, LT, LB
```

The L registers are required to be zero on entry and exit from a function. A user may specify that a function clobbers the L registers. If it is a compiler-generated function, then it leaves the L registers zero at the end of the function. If it is an assembly function, it may clobber the L registers. In that case, the L registers are re-zeroed after any call to that function.

The SEQSTAT, RETI, RETX, RETN, SYSCFG, CYCLES, and CYCLES2 registers are never used by the compiler and are never preserved.

When building for pre-Blackfin+ processors, Register P1 is used by the linker to convert out-of-range jump and call instructions into indirect jumps and calls, so P1 may be modified at the call site regardless of whether the regs_clobbered pragma says it is clobbered. Therefore, when building for these processors, the compiler never keeps P1 live across a call. However, the compiler accepts the pragma when compiling a function in case you want to keep P1 live across a call that is not expanded by the linker. It is your responsibility to make sure such calls are not expanded by the linker.

Conversion into indirect jump and calls via P1 does not happen for Blackfin+ processors, which support direct jump and call instructions that can reach the entire address range.

User-Reserved Registers

User-reserved registers, indicated via the -reserve *register[, register]* switch are never preserved in the function wrappers, whether in the clobbered set or not.

Function Parameters

Function calling conventions are visible to the caller and do not affect the clobbered set that may be used on a function.

In the following example, the parameters `a` and `b` are passed in registers `R0` and `R1`, respectively. No matter what happens in function `f`, after the call returns, the values of `R0` and `R1` remain 2 and 3, respectively.

```
#pragma regs_clobbered ""   // clobbers nothing
void f(int a, int b);
void g() {
   f(2,3);
}
```

Function Results

The registers in which a function returns its result must always be clobbered by the callee and retain their new value in the caller. They may appear in the clobbered set of the callee, but it does not matter to the generated code-the return register are not saved and restored. Only the return register used by the particular function return type is special. Return registers used by different return types are treated in the clobbered list in the conventional way.

For example,

```
typedef struct { int x; int y; } Point;
typedef struct { int x[10]; } Big;
int f();          // Result in R0.
                  // R1, P0 may be preserved across call.
Point g();        // Result in R0 and R1.
                  // P0 may be preserved across call.
Big f();          // Result pointer in P0.
                  // R0, R1 may be preserved across call.
```

#pragma regs_clobbered_call *string*

The `regs_clobbered_call` pragma may be applied to a statement to indicate that the call within the statement uses a modified volatile register set. The pragma is closely related to `#pragma regs_clobbered`, but avoids some of the restrictions that relate to that pragma.

These restrictions arise because the `regs_clobbered` pragma applies to a function's declaration-when the call is made, the clobber set is retrieved from the declaration automatically. This is not possible when the declaration is not available, because the function being called is not directly tied to a declaration of a specific function. This affects:

- Pointers to functions
- Class methods
- Pointers to class methods

- Virtual functions

In such cases, the `regs_clobbered_call` pragma can be used at the call site to inform the compiler directly of the volatile register set to be used during the call.

The pragma's syntax is as follows:

```
#pragma regs_clobbered_call "clobber_string"
statement
```

where *clobber_string* follows the same format as the `regs_clobbered` pragma, and *statement* is the C statement containing the call expression.

There must be only a single call within the statement; otherwise, the statement is ambiguous.

For example,

```
#pragma regs_clobbered "r0 r1 p1"
int func(int arg) { /* some code */ }
int (*fnptr)(int) = func;
int caller(int value) {
   int r;
   #pragma regs_clobbered_call "r0 r1"
   r = (*fnptr)(value);
   return r;
}
```

> **NOTE:** When using the `regs_clobbered_call` pragma, ensure that the called function does indeed only modify the registers listed in the clobber set for the call—the compiler does not check this for you. It is valid for the callee to clobber fewer registers than those listed in the call's clobber set. It is also valid for the callee to modify registers outside of the call's clobber set, as long as the callee saves the values first and restores them before returning to the caller.

The following examples show this.

Example 1:

```
#pragma regs_clobbered "r0 r1"
void callee(void) { ... }

#pragma regs_clobbered_call "r0 r1"
callee();        // Okay - clobber sets match
```

Example 2:

```
#pragma regs_clobbered "r0"
void callee(void) { ... }

#pragma regs_clobbered_call "r0 r1"
callee();        // Okay - callee clobber set is a subset
                 // of call's set.
```

Example 3:

```
#pragma regs_clobbered "r0 r1 r2"
void callee(void) { ... }

#pragma regs_clobbered_call "r0 r1"
callee();        // Error - callee clobbers more than
                 // indicated by call.
```

Example 4:

```
void callee(void) { ... }

#pragma regs_clobbered_call "r0 r1"
callee();        // Error - callee uses default set larger
                 // than indicated by call.
```

Limitations

Pragma `regs_clobbered_call` may not be used on constructors or destructors of C++ classes.

The pragma only applies to the call in the immediately-following statement. If the immediately-following line contains more than one statement, the pragma only applies to the first statement on the line:

```
#pragma regs_clobbered_call "r0 r1"
x = foo(); y = bar();   // only "x = foo();" is affected
                        // by the pragma.
```

Similarly, if the immediately-following line is a sequence of declarations that use calls to initialize the variables, only the first declaration is affected:

```
#pragma regs_clobbered_call "r0 r1"
int x = foo(), y = bar();   // only "x = foo()" is affected
                            // by the pragma.
```

Moreover, if the declaration with the call-based initializer is not the first in the declaration list, the pragma will have no effect:

```
#pragma regs_clobbered_call "r0 r1"
int w = 4, x = foo(); y = bar();   // pragma has no effect
                                   // on "w = 4".
```

The pragma has no effect on function calls that get inlined. Once a function call is inlined, the inlined code obeys the clobber set of the function into which it has been inlined. It does not continue to obey the clobber set that will be used if an out-of-line copy is required.

#pragma overlay

When compiling code that involves one function calling another in the same source file, the compiler optimizer can propagate register information between the functions. This means that it can record which scratch registers are clobbered over the function call. This can cause problems when compiling overlaid functions, as the compiler may assume that certain scratch registers are not clobbered over the function call, but they are clobbered by the overlay manager. The #pragma overlay, when placed on the definition of a function, will disable this propagation of register information to the function's callers.

For example,

```
#pragma overlay
int add(int a, int b)
{
   // callers of function add() assume it clobbers
   // all scratch registers
   return a+b;
}
```

#pragma result_alignment (*n*)

The `result_alignment` pragma asserts that the pointer or integer returned by the function has a value that is a multiple of *n*. The pragma is often used in conjunction with the `#pragma alloc` of custom-allocation functions that return pointers more strictly aligned than could be deduced from their type. This pragma has no effect on processors with Blackfin+ architecture.

Format Argument Pragmas

The format argument pragmas `__printf_args` and `__scanf_args` can be used before a function declaration to tell the compiler to warn about mismatches between format string and variable arguments in calls to the function.

#pragma __printf_args

The `__printf_args` pragma can be placed before a function prototype or definition. It tells the compiler that the function interprets its arguments like the standard `printf` family of functions, which take a format string argument followed by a variable-length argument list corresponding to specifiers in the format string. When such a function is called, the compiler checks that the number and types of variable arguments match the specifiers in the format string, issuing a warning if they do not.

The pragma is ignored for functions that do not take a string argument followed by a variable-length argument list.

For example,

```
#pragma __printf_args
void error(char *where, char *msg, ...);
```

With that, the following invocation of the function will result in a warning that the format string requires an additional argument:

```
error("Parser", "Unexpected token '%s'");
```

#pragma __scanf_args

The `__scanf_args` pragma can be placed before a function prototype or definition. It tells the compiler that the function interprets its arguments like the standard `scanf` family of functions, which take a format string argument followed by a variable-length argument list corresponding to specifiers in the format string. When such a function is called, the compiler checks that the number and types of variable arguments match the specifiers in the format string, issuing a warning if they do not.

The pragma is ignored for functions that do not take a string argument followed by a variable-length argument list.

For example:
```
#pragma __scanf_args
void scan_channel(int chan, char *format, ...);
```

With that, the following invocation of the function will result in a warning that the format string requires an additional argument:
```
scan_channel(3, "%u,%u", &x);
```

Class Conversion Optimization Pragmas

The class conversion optimization pragmas (param_never_null and suppress_null_check) allow the compiler to generate more efficient code when converting class pointers from a pointer-to-derived-class to a pointer-to-base-class, by asserting that the pointer to be converted will never be a null pointer. This allows the compiler to omit the null check during conversion.

#pragma param_never_null *param_name* [...]

The param_never_null pragma must immediately precede a function definition. It specifies a name or a list of space-separated names, which must correspond to the parameter names declared in the function definition. It checks that the named parameter is a class pointer type. Using this information allows it to generate more efficient code for a conversion from a pointer to a derived class to a pointer to a base class. It removes the need to check for the null pointer during the conversion. For example,

```
#include <iostream>

using namespace std;
class A {
    int a;
};
class B {
    int b;
};
class C: public A, public B {
    int c;
};

C obj;
B *bpart = &obj;
bool fail = false;

#pragma param_never_null pc
void func(C *pc)
{
   B *pb;
   pb = pc;     /* without pragma the code generated has to check for NULL */
   if (pb != bpart)
      fail = true;
}

int main(void)
```

```
{
   func(&obj);
   if (fail)
      cout << "Test failed" << endl;
   else
       cout << "Test passed" << endl;
   return 0;
}
```

#pragma suppress_null_check

The `suppress_null_check` pragma must immediately precede an assignment of two pointers or a declaration list. If the pragma precedes an assignment, it indicates that the second operand pointer is not null and generates more efficient code for a conversion from a pointer to a derived class to a pointer to a base class. It removes the need to check for the null pointer before assignment. On a declaration list, it marks all variables as not being the null pointer. If the declaration contains an initialization expression, that expression is not checked for null.

```
#include <iostream>

using namespace std;
class A {
    int a;
};
class B {
    int b;
};
class C: public A, public B {
    int c;
};

C obj;
B *bpart = &obj;
bool fail = false;

void func(C *pc)
{
   B *pb;
   #pragma suppress_null_check
   pb = pc;      /* without pragma the code generated has to
                     check for NULL */
   if (pb != bpart)
       fail = true;
}

void func2(C *pc)
{
    #pragma suppress_null_check
    B *pb = pc, *pb2 = pc;
                      /* pragma means these initializations need not check for
                         NULL. It also marks pb and pb2 as never being NULL,
```

```
                        so the compiler will not generate NULL checks in class
                        conversions using these pointers. */
    if (pb != bpart || pb2 != bpart)
        fail = true;
}

int main(void)
{
    func(&obj);
    func2(&obj);
    if (fail)
        cout << "Test failed" << endl;
    else
        cout << "Test passed" << endl;
    return 0;
}
```

Template Instantiation Pragmas

The template instantiation pragmas (`instantiate`, `do_not_instantiate`, and `can_instantiate`) provide fine-grained control over where (that is, in which object file) the individual instances of template functions, member functions, and static members of template classes are created. The creation of these instances from a template is known in *C++ speak* as instantiation. As templates are a feature of C++, these pragmas are allowed only in C++ mode.

Refer to Compiler C++ Template Support for more information on how the compiler handles templates.

The instantiation pragmas take the name of an instance as a parameter, as shown in the *Instance Names* table.

Table 2-38: Instance Names

Name	*Parameter*
Template class name	`A<int>`
Template class declaration	`class A<int>`
Member function name	`A<int>::f`
Static data member name	`A<int>::I`
Static data declaration	`int A<int>::I`
Member function declaration	`void A<int>::f(int, char)`
Template function declaration	`char* f(int, float)`

If the instantiation pragmas are not used, the compiler selects object files where all required instances automatically instantiate during the prelinking process.

#pragma instantiate *instance*

The `instantiate` pragma requests the compiler to instantiate *instance* in the current compilation.

The following example causes all static members and member functions for the `int` instance of a template class `Stack` to be instantiated, whether they are required in this compilation or not.

```
#pragma instantiate class Stack<int>
```

The following example causes only the individual member function `Stack<int>::push(int)` to be instantiated.

```
#pragma instantiate void Stack<int>::push(int)
```

#pragma do_not_instantiate *instance*

The `do_not_instantiate` pragma directs the compiler not to instantiate *instance* in the current compilation.

The following example prevents the compiler from instantiating the static data member `Stack<float>::use_count` in the current compilation.

```
#pragma do_not_instantiate int Stack<float>::use_count
```

#pragma can_instantiate *instance*

The `can_instantiate` pragma tells the compiler that if *instance* is required anywhere in the program, it should be instantiated in this compilation.

> **NOTE:** Currently, this pragma forces the instantiation, even if it is not required anywhere in the program. Therefore, it has the same effect as `#pragma instantiate`.

Header File Control Pragmas

The header file control pragmas (`no_implicit_inclusion`, `once`, and `system_header`) help the compiler to handle header files.

#pragma no_implicit_inclusion

With the -c++ switch, for each included header file (`.h` or non-suffixed), the compiler attempts to include the corresponding `.c` or `.cpp` file. This is called "*implicit inclusion*".

If `#pragma no_implicit_inclusion` is placed in an `.h` (or non-suffixed) file, the compiler does not implicitly include the corresponding `.c` or `.cpp` file with the `-c++` switch. This behavior only affects the `.h` (or non-suffixed) file with `#pragma no_implicit_inclusion` within it and the corresponding `.c` or `.cpp` files.

For example, if there are the following files,

`t.c` containing

```
#include "m.h"
```

and `m.h` and `m.c` are both empty, then

```
ccblkfn -c++ t.c -M
```

shows the following dependencies for `t.c`:

```
t.doj: t.c
t.doj: m.h
t.doj: m.c
```

If the following line is added to `m.h`,

```
#pragma no_implicit_inclusion
```

running the compiler as before would not show `m.c` in the dependencies list, such as:

```
t.doj: t.c
t.doj: m.h
```

#pragma once

The `once` pragma, which should appear at the beginning of a header file, tells the compiler that the header is written in such a way that including it several times has the same effect as including it once.

For example,

```
#pragma once
#ifndef FILE_H
   #define FILE_H
   ... contents of header file ...
#endif
```

> **NOTE:** In this example, `#pragma once` is actually optional because the compiler recognizes the `#ifndef`, `#define`, or `#endif` idioms and does not reopen a header that uses it.

#pragma system_header

The `system_header` pragma identifies an include file as a file supplied with CCES. The CCES compiler uses this information to help optimize uses of the supplied library functions and inline functions that these files define. Do not use this pragma in user application source.

Diagnostic Control Pragmas

The compiler supports `#pragma diag`, which allows selective modification of the severity of compiler diagnostic messages.

The directive has these forms:

- Modify the severity of specific diagnostics
- Modify the behavior of an entire class of diagnostics
- Save or restore the current behavior of all diagnostics

Modifying the Severity of Specific Diagnostics

This form of the directive has the following syntax:

```
#pragma diag(ACTION: DIAG[, DIAG ...][: STRING])
```

The `action:` qualifier can be one of the keywords in the *Keywords for ACTION Qualifier* table.

Table 2-39: Keywords for ACTION Qualifier

Keyword	Action
suppress	Suppresses all instances of the diagnostic
remark	Changes the severity of the diagnostic to a remark
annotation	Changes the severity of the diagnostic to an annotation
warning	Changes the severity of the diagnostic to a warning
error	Changes the severity of the diagnostic to an error
restore	Restores the severity of the diagnostic to what it was originally at the start of compilation after all command-line options were processed

If not in MISRA-C mode, the *DIAG* qualifier can be one or more comma-separated compiler diagnostic message numbers without any preceding "cc" or zeros. The choice of error numbers is limited to those that may have their severity overridden (such as those that display "{D}" in the error message).

In addition, some diagnostics are *global* (for example, diagnostics emitted by the compiler back-end after lexical analysis and parsing, or before parsing begins), and these global diagnostics cannot have their severity overridden by the diagnostic control pragmas. To modify the severity of global diagnostics, use the diagnostic control switches. For more information, see -W{annotation|error|remark|suppress|warn} *number[, number...]*.

In MISRA-C mode, the *DIAG* qualifier is a list of MISRA-C rule numbers in the form `misra_rule_6_3` and `misra_rule_19_4` for rules 6.3 and 19.4, and so on. Rules 10.1 and 10.2 are a special case, in which both rules split into four distinct rule checks. For example, 10.1(c) should be stated as `misra_rule_10_1_c`. *DIAG* may also be the special token misra_rules_all, which specifies that the pragma applies to all MISRA-C rules.

The third optional argument is a string-literal to insert a comment regarding the use of the `#pragma diag`.

Modifying the Behavior of an Entire Class of Diagnostics

This form of the directive has the following syntax, which is not allowed in MISRA-C mode:

```
#pragma diag( ACTION )
```

The effects are as follows:

- `#pragma diag(errors)`

  The pragma can be used to inhibit all subsequent warnings and remarks (equivalent to the -w switch).

- `#pragma diag(remarks)`

  The pragma can be used to enable all subsequent remarks, annotations and warnings (equivalent to the -Wremarks switch).

- `#pragma diag(annotations)`

  The pragma can be used to enable all subsequent annotations and warnings (equivalent to the -Wannotations switch).

- `#pragma diag(warnings)`

The pragma can be used to restore the default behavior when the `-w`, `-Wremarks`, and `-Wannotations` switches are not specified, which is to display warnings but inhibit remarks and annotations.

Saving or Restoring the Current Behavior of All Diagnostics

This form has the following syntax:

```
#pragma diag(ACTION)
```

The effects are as follows:

- `#pragma diag(push)`

  The pragma may be used to store the current state of the severity of all diagnostic error messages.

- `#pragma diag(pop)`

  The pragma restores all diagnostic error messages that were previously saved with the most recent `push`.

All `#pragma diag(push)` directives must be matched with the same number of `#pragma diag(pop)` directives in the overall translation unit, but need not be matched within individual source files, unless in MISRA-C mode. Note that the error threshold (set by the `remarks`, `annotations`, `warnings`, or `errors` keywords) is also saved and restored with these directives.

The duration of such modifications to diagnostic severity are from the next line following the pragma to the end of the translation unit, the next `#pragma diag(pop)` directive, or the next overriding `#pragma diag()` directive with the same error number. These pragmas may be used anywhere and are not affected by normal scoping rules.

All command-line overrides to diagnostic severity are processed first, and any subsequent `#pragma diag()` directives take precedence, with the restore action changing the severity back to that at the start of compilation after processing the command-line switch overrides.

NOTE: Directives to modify specific diagnostics are singular (for example, "error"), and the directives to modify classes of diagnostics are plural (for example, "errors").

Run-Time Checking Pragmas

Run-time checking pragmas allow you to control the compiler's generation of additional checking code. This code can test at run-time for common programming errors. The `-rtcheck` and its related switches control which common errors are tested for. Use the command-line switches to enable run-time checking; once run-time checking is enabled, the run-time checking pragmas can be used to disable and re-enable checking, for specific functions. The -rtcheck and its related switches control which common errors are tested for. Use the command-line switches to enable run-time checking; once run-time checking is enabled, the run-time checking pragmas can be used to disable and re-enable checking, for specific functions.

This section describes the following pragmas:

- #pragma rtcheck(off)

- #pragma rtcheck(on)

NOTE: Run-time checking causing the compiler to generate additional code to perform the checks. This code has space and performance overheads. Use of run-time checking should be restricted to application development, and should not be used on applications for release.

#pragma rtcheck(off)

The `rtcheck(off)` pragma disables any run-time check code generation that has been enabled via command-line switches such as -rtcheck. The pragma is only valid at file scope, and affects code generation for function definitions that follow.

The pragma has no effect on checks of heap operations. This is because such checks are provided by selecting alternative library support at link-time, and so apply to the whole application.

#pragma rtcheck(on)

The `rtcheck(on)` pragma re-enables any run-time check code generation that was enabled via command-line switches such as -rtcheck. The pragma is only valid at file scope, and affects code generation for function definitions that follow. If no run-time checking was enabled by command-line switches, the pragma has no effect.

Memory Bank Pragmas

The memory bank pragmas provide additional performance characteristics for the memory areas used to hold code and data for the function.

By default, the compiler assumes that there are no external costs associated with memory accesses. This strategy allows optimal performance when the code and data are placed into high-performance internal memory. In cases where the performance characteristics of memory are known in advance, the compiler can exploit this knowledge to improve the scheduling of generated code.

#pragma code_bank(*bankname*)

The `code_bank` pragma informs the compiler that the instructions for the immediately-following function are placed in a memory bank called *bankname*. Without this pragma, the compiler assumes that instructions are placed into the default bank, if one has been specified; see Memory Bank Selection for details. When optimizing the function, the compiler is aware of attributes of memory bank *bankname*, and determines how long it takes to fetch each instruction from the memory bank.

If *bankname* is omitted, the instructions for the function are not considered to be placed into any particular bank.

In the following example, the `add_slowly()` function is placed into the `slowmem` bank, which may have different performance characteristics from the default code bank, into which `add_quickly()` is placed.

```
#pragma code_bank(slowmem)
int add_slowly (int x, int y) { return x + y; }
int add_quickly(int a, int b) { return a + b; }
```

#pragma data_bank(*bankname*)

The `data_bank` pragma informs the compiler that the immediately-following function uses the memory bank *bankname* as the model for memory accesses for non-local data that does not otherwise specify a memory bank; see Memory Bank Selection for details. Without this pragma, the compiler assumes that non-local data should use the default bank, if any has been specified, for behavioral characteristics.

If *bankname* is omitted, the non-local data for the function is not considered to be placed into any specific bank.

In both `green_func()` and `blue_func()` of the following example, `i` is associated with the memory bank "`blue`", and the retrieval and update of `i` are optimized to use the performance characteristics associated with memory bank "`blue`".

```
#pragma data_bank(green)
int green_func(void)
{
   extern int arr1[32];
   extern int bank("blue") i;
   i &= 31;
   return arr1[i++];
}
int blue_func(void)
{
   extern int arr2[32];
   extern int bank("blue") i;
   i &= 31;
   return arr2[i++];
}
```

The array `arr1` does not have an explicit memory bank in its declaration. Therefore, it is associated with the memory bank "`green`", because `green_func()` has a specific default data bank. In contrast, `arr2` is associated with the default data memory bank (if any), because `blue_func()` does not have a `#pragma data_bank` preceding it.

#pragma stack_bank(*bankname*)

The `stack_bank` pragma informs the compiler that all locals for the immediately-following function are to be associated with memory bank *bankname*, unless they explicitly identify a different memory bank. Without this pragma, all locals are assumed to be associated with the default stack memory bank, if any; see Memory Bank Selection for details.

If *bankname* is omitted, locals for the function are not considered to be placed into any particular bank.

In the following example, the `dotprod()` function places the `sum` and `i` values into memory bank `mystack`, while `fib()` places `r`, `a`, and `b` into the default stack memory bank (if any), because there is no `stack_bank` pragma. The `count_ticks()` function does not declare any local data, but any compiler-generated local storage uses the `sysstack` memory bank's performance characteristics.

```
#pragma stack_bank(mystack)
short dotprod(int n, const short *x, const short *y)
{
   int sum = 0;
   int i = 0;
   for (i = 0; i < n; i++)
      sum += *x++ * *y++;
   return sum;
}
int fib(int n)
```

```
{
   int r;
   if (n  2) {
      r = 1;
    } else {
       int a = fib(n-1);
       int b = fib(n-2);
       r = a + b;
    }
    return r;
}
#pragma stack_bank(sysstack)
void count_ticks(void)
{
   extern int ticks;
   ticks++;
}
```

#pragma default_code_bank(bankname)

The `default_code_bank` pragma informs the compiler that *bankname* should be considered the default memory bank for the instructions generated for any following functions that do not explicitly use `#pragma code_bank`.

If *bankname* is omitted, the pragma sets the compiler's default back to not specifying a particular bank for generated code.

For more information, see Memory Bank Selection.

#pragma default_data_bank(bankname)

The `default_data_bank` pragma informs the compiler that *bankname* should be considered the default memory bank for non-local data accesses in any following functions that do not explicitly use `#pragma data_bank`.

If *bankname* is omitted, the pragma sets the compiler's default back to not specifying a particular bank for non-local data.

For more information, see Memory Bank Selection.

#pragma default_stack_bank(bankname)

The `default_stack_bank` pragma informs the compiler that *bankname* should be considered the default memory bank for local data in any following functions that do not explicitly use `#pragma stack_bank`.

If *bankname* is omitted, the pragma sets the compiler's default back to not specifying a particular bank for local data.

For more information, see Memory Bank Selection.

#pragma bank_memory_kind(*bankname, kind*)

The `bank_memory_kind` pragma informs the compiler of what *kind* of memory the memory bank *bankname* is. See Memory Bank Kinds for the kinds supported by the compiler.

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition.

In the following example, the compiler knows that all accesses to the `data[]` array are to the "`blue`" memory bank, and hence to internal, in-core memory.

```
#pragma bank_memory_kind(blue, internal)
int sum_list(const int bank("blue") *data, int n)
{
   int sum = 0;
   while (n--)
      sum += data[n];
   return sum;
}
```

#pragma bank_read_cycles(*bankname, cycles[, bits]*)

The `bank_read_cycles` pragma tells the compiler that each read operation on the memory bank *bankname* requires *cycles* cycles before the resulting data is available. This allows the compiler to generate more efficient code.

If the `bits` parameter is specified, it indicates that a read of *bits* bits will take *cycles* cycles. If the *bits* parameter is omitted, the pragma indicates that reads of all widths will require *cycles* cycles. *bits* may be one of 8, 16 or 32.

In the following example, the compiler assumes that a read from `*x` takes a single cycle, as this is the default read time, but that a read from `*y` takes twenty cycles, because of the pragma.

```
#pragma bank_read_cycles(slowmem, 20)
int dotprod(int n, const int *x, bank("slowmem") const int *y)
{
   int i, sum;
   for (i=sum=0; i < n; i++)
      sum += *x++ * *y++;
   return sum;
}
```

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition.

#pragma bank_write_cycles(*bankname, cycles[, bits]*)

The `bank_write_cycles` pragma tells the compiler that each write operation on memory bank *bankname* requires *cycles* cycles before it completes. This allows the compiler to generate more efficient code.

If the `bits` parameter is specified, it indicates that a write of *bits* bits will take *cycles* cycles. If the *bits* parameter is omitted, the pragma indicates that writes of all widths will require *cycles* cycles. *bits* may be one of 8, 16 or 32.

In the following example, the compiler knows that each write through `ptr` to the "`output`" memory bank takes six cycles to complete.

```
#pragma bank_write_cycles(output, 6)
void write_buf(int n, const char *buf)
{
   volatile bank("output") char *ptr = REG_ADDR;
   while (n--)
      *ptr = *buf++;
}
```

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition

#pragma bank_maximum_width(*bankname*, *width*)

The `bank_maximum_width` pragma informs the compiler that *width* is the maximum number of bits to transfer to/from memory bank *bankname* in a single access. On Blackfin processors, the *width* parameter may only be 32.

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition.

Exceptions Tables Pragma

`#pragma generate_exceptions_tables`

The `generate_exceptions_tables` pragma may be applied to a C function definition to request the compiler to generate tables that enable C++ exceptions to be thrown through executions of this function.

This example consists of two source files. The first is a C file that contains the pragma applied to the definition of function `call_a_call_back`.

```
#pragma generate_exceptions_tables
void call_a_call_back(void pfn(void)) {
   pfn();   /* without pragma program terminates
            when throw_an_int throws an exception */
}
```

The second source file contains C++ code. The function `main` calls `call_a_call_back`, from the C file listed above, which in turn calls `throw_an_int`. The exception thrown by `throw_an_int` will be caught by the catch handler in `main` because use of the pragma ensured the compiler generated an exceptions table for `call_a_call_back`.

```
#include <iostream>

extern "C" void call_a_call_back(void pfn());
static void throw_an_int() {
```

```
    throw 3;
}   int main() {
      try {
    call_a_call_back(throw_an_int);
  } catch (int i) {
    if (i == 3) std::cout  "Test passed\n";
  }
 }
```

An alternative to using `#pragma generate_exceptions_tables` is to compile C files with the -eh (enable exception handling) switch which, for C files, is equivalent to using the pragma before every function definition.

GCC Compatibility Extensions

The compiler provides compatibility with many features of the C dialect accepted by version 4.6 of the GNU C compiler. Many of these features are available in the ISO/IEC 9899:1999 C standard. A brief description of the extensions is included in this section. For more information, refer to the following web address: `http://gcc.gnu.org`.

>   **NOTE:** GCC compatibility extensions not available in C++ mode, unless enabled with the -g++ switch.

Statement Expressions

A *statement expression* is a compound statement enclosed in parentheses. Because a compound statement itself is enclosed in braces as `{ }`, this construct is enclosed in parentheses-brace pairs, as `({ })`.

The value computed by a statement expression is the value of the last statement (which should be an expression statement). The statement expression may be used where expressions of its result type may be used. But they are not allowed in constant expressions.

Statement expressions are useful in the definition of macros as they allow the declaration of variables local to the macro.

In the following example, the `foo()` and `thing()` statements get called once each because they are assigned to the variables `__x` and `__y`, which are local to the statement expression that `min` expands to. The `min()` can be used freely within a larger expression because it expands to an expression.

```
#define min(a,b) ({                            \
   short __x=(a),__y=(b),__res;                \
   if (__x > __y)                              \
       __res = __y;                            \
   else                                        \
       __res = __x;                            \
       __res;                                  \
   })

   int use_min() {
   return  min(foo(), thing()) + 2;
}
```

Labels local to a statement expression can be declared with the `__label__` keyword. For example,

```
#define checker(p)       ({           \
    __label__ exit;                    \
    int i;                             \
    for (i=0; p[i]; ++i) {             \
        int d = get(p[i]);             \
        if (!check(d)) goto exit;      \
          process(d);                  \
      }                                \
      exit:                            \
      i;                               \
})

extern int g_p[100];
int checkit()   {
   int local_i = checker(g_p);
   return local_i;
}
```

NOTE: Statement expressions are not supported in C++ mode unless GNU C++ extensions are enabled using the -g++ switch. Statement expressions are an extension to C originally implemented in the GCC compiler. Analog Devices supports the extension primarily to aid porting code written for that compiler. When writing new code, consider using inline functions, which are compatible with ANSI/ISO standard C++ and C99, and are as efficient as macros when optimization is enabled.

Type Reference Support Keyword (typeof)

The `typeof(expression)` construct can be used as a name for the type of expression without actually knowing what that type is. It is useful for making source code that is interpreted more than once, such as macros or include files, more generic. The `typeof` keyword may be used wherever a `typedef` name is permitted such as in declarations and in casts.

The following example shows `typeof` used in conjunction with a statement expression to define a "generic" macro with a local variable declaration.

```
#define abs(a)  ({                \
    typeof(a) __a = a;            \
    if (__a  0) __a = - __a;      \
    __a;                          \
})
```

The argument to `typeof` may also be a type name. Because `typeof` itself is a type name, it may be used in another `typeof( type-name )` construct. This can be used to restructure the C-type declaration syntax.

The following example declares `y` to be an array of four pointers to `char`.

```
#define pointer(T)    typeof(T *)
#define array(T, N)   typeof(T [N])

array (pointer (char), 4) y;
```

NOTE: The `typeof` keyword is not supported in C++ mode unless GNU C++ extensions are enabled using the -g++ switch. The `typeof` keyword is an extension to C originally implemented in the GCC compiler. It should be used with caution because it is not compatible with other dialects of C/C++ and has not been adopted by the more recent C99 standard.

Conditional Expressions With Missing Operands

The middle operand of a conditional operator can be omitted. If the condition is nonzero (true), the condition itself is the result of the expression. This can be used for testing and substituting a different value when a pointer is NULL. The condition is evaluated only once; therefore, repeated side effects can be avoided.

This is an extension to C, provided for compatibility with GCC. It is not allowed in C++ mode unless GNU C++ extensions are enabled using the -g++ switch.

The following example calls `lookup()` once, and substitutes the string "-" if it returns NULL.

```
printf("name = %s\n", lookup(key)?:"-");
```

Zero-Length Arrays

Arrays may be declared with zero length. This anachronism is supported to provide compatibility with GCC. Use variable-length array members instead.

GCC Variable Argument Macros

The final parameter in a macro declaration may be followed by dots (...) to indicate the parameter stands for a variable number of arguments.

For example,

```
#define tracegcc(file,line,msg ...) \
            logmsg(file,line, ## msg)
```

can be used with differing numbers of arguments: the following statements:

```
tracegcc("a.c", 999, "one", "two", "three");
tracegcc("a.c", 999, "one", "two");
tracegcc("a.c", 999, "one");
tracegcc("a.c", 999);
```

expand to the following code:

```
logmsg("a.c", 999,"one", "two", "three");
logmsg("a.c", 999,"one", "two");
logmsg("a.c", 999,"one");
logmsg("a.c", 999);
```

The `##` operator has a special meaning when used in a macro definition before the parameter that expands the variable number of arguments: if the parameter expands to nothing, it removes the preceding comma.

NOTE: The variable argument macro syntax comes from GCC. The compiler supports GCC variable argument macro formats in C89 and C99 modes by default, and in C++ mode when the -g++ switch is enabled. In addition, see Variable Argument Macros.

Line Breaks in String Literals

String literals may span many lines. The line breaks do not need to be escaped in any way. They are replaced by the character \n in the generated string. This extension is not supported in C++ mode unless GNU C++ extensions are enabled using the -g++ switch. The extension is not compatible with many dialects of C, including ISO IEC 9899:1990 and ISO/IEC 9899:1999. However, it is useful in asm statements, which are intrinsically non-portable.

This extension may be disabled via the switch -no-multiline.

Arithmetic on Pointers to Void and Pointers to Functions

Addition and subtraction is allowed on pointers to void and pointers to functions. The result is as if the operands had been cast to pointers to char. The sizeof operator returns one for void and function types.

This extension is not available in C++ mode unless GNU C++ extensions are enabled with the -g++ switch.

Cast to Union

A type cast can be used to create a value of a union type, by casting a value of one of the union's member types to the union type.

This is not supported in C++ mode.

Ranges in Case Labels

A consecutive range of values can be specified in a single case by separating the first and last values of the range with the three-period token "...".

For example,

```
case 200 ... 300:
```

This is not supported in C++ mode unless GNU C++ extensions are enabled using the -g++ switch.

Escape Character Constant

The escape character \e may be used in character and string literals. It maps to the ASCII Escape code, 27.

This extension is not available in C++ mode unless GNU C++ extensions are enabled using the -g++ switch.

Alignment Inquiry Keyword (__alignof__)

The __alignof__ (*type-name*) construct evaluates to the alignment required for an object of a type. The __alignof__ *expression* construct can also be used to give the alignment required for an object of the *expression* type.

If an *expression* is an lvalue (may appear on the left side of an assignment), the returned alignment takes into account alignment requested by pragmas and the default variable allocation rules.

(asm) Keyword for Specifying Names in Generated Assembler

The asm keyword can be used to direct the compiler to use a different name for a global variable or function. See also #pragma linkage_name *identifier* .

This extension is not available in C++ mode unless GNU C++ extensions are enabled using the -g++ switch.

The following example instructs the compiler to use the label `C11045` in the assembly code it generates wherever it needs to access the source level variable `N`. By default, the compiler uses the label `_N`.

```
int N asm("C11045");
```

The `asm` keyword can also be used in function declarations, but not in function definitions. However, a definition preceded by a declaration has the desired effect. For example,

```
extern int f(int, int) asm("func");
int f(int a, int b) {
    ...
}
```

Function, Variable, and Type Attribute Keyword (__attribute__)

The `__attribute__` keyword can be used to specify attributes of functions, variables, and types, as in the following examples:

```
void func(void) __attribute__ ((section("fred")));
int a __attribute__ ((aligned (8)));
typedef struct {int a[4];} __attribute__((aligned (4))) Q;
```

Support for the `__attribute__` keyword means that fewer changes may be required when porting GCC code. This extension is not available in C++ mode unless enabled using the -g++ switch.

The *Keywords for __attribute__* table lists the accepted keywords.

Table 2-40: Keywords for __attribute__

Attribute Keyword	Description
alias(*"name"*)	Accepted on functions declarations. Declares the function to be an alias for *name*.
aligned(*N*)	Accepted on variables, where it is equivalent to `#pragma align(N)`. Accepted (but ignored) on typedefs.
always_inline	Accepted on function declarations. Equivalent to the pragma of the same name.
const	Accepted on function declarations. Equivalent to the pragma of the same name.
constructor	Accepted (but ignored) on function declarations.
deprecated	Accepted on function, variable and type declarations. Causes the compiler to emit a warning if the entity with the attribute is referenced within the source code.
destructor	Accepted (but ignored) on function declarations.
format(*kind,str,args*)	Accepted on function declarations. Indicates that the function accepts a formatting argument string of type *kind*, e.g. `printf`. *str* and *args* are integer values; the *str*th parameter of the function is the formatting string, while the *args*th parameter of the function is the first parameter processed by the formatting string.

Table 2-40: Keywords for __attribute__ (Continued)

Attribute Keyword	Description
format_arg(*kind, str*)	Accepted on function declarations. Indicates that the function accepts and returns a formatting argument string of type *kind*. *str* is an integer value; the *str*th parameter of the function is the formatting string.
malloc	Accepted on function declarations. Equivalent to using #pragma alloc.
naked	Accepted (but ignored) on function declarations.
no_instrument_function	Accepted (but ignored) on function declarations.
nocommon	Accepted on variable declaration. Ignored when (-decls-{weak|strong}) is in effect. Makes a declaration strong when (-decls-{weak|strong}) is in effect.
noinline	Accepted on function declarations. Equivalent to #pragma never_inline.
nonnull	Accepted on function declarations. Causes the compiler to emit a warning if the function is invoked with any NULL parameters.
noreturn	Accepted on function declarations. Equivalent to using the pragma of the same name.
nothrow	Accepted (but ignored) on function declarations.
packed	Accepted (but ignored) on typedefs. When used on variable declarations, this is equivalent to using the pragma of the same name.
pure	Accepted on function declarations. Equivalent to using the pragma of the same name.
section("*name*")	Accepted on function declarations. Equivalent to using the pragma of the same name.
sentinel	Accepted on function declarations. Directs the compiler to emit a warning for any calls to the function which do not provide a null pointer literal as the last parameter. Accepts an optional integer position *P* (default 0) to indicate that the *P*th parameter from the end is the sentinel instead.
transparent_union	Accepted on union definitions. When the union type is used for a function's parameter, the parameter can accept values which match any of the union's types.
unused	Accepted on declarations of functions, variables and types. Indicates that the entity is known not to be used, so the compiler should not emit diagnostics complaining that there are no uses of the entity.
used	Accepted on declarations of functions and variables. Indicates that the compiler should emit the entity even when the compiler cannot detect uses. Similar to #pragma retain_name, but this attribute can be applied to static entities that will not be visible outside the module. Conversely, this attribute will not prevent linker elimination from deleting the entity.
warn_unused_result	Accepted (but ignored) on function declarations.
weak	Accepted on function and variable declarations. Equivalent to using #pragma weak_entry

Unnamed struct/union Fields Within struct/unions

The compiler allows you to define a structure or union that contains, as fields, structures and unions without names. For example,

```
struct {
    int field1;
```

```
    union {
    int field2;
    int field3;
};
int field4;
myvar;
```

This allows you to access the members of the unnamed union as though they were members of the enclosing struct or union, for example, `myvar.field2`.

Preprocessor-Generated Warnings

The preprocessor directive `#warning` causes the preprocessor to generate a warning and continue preprocessing. The text that follows the `#warning` directive on the line is used as the warning message. For example,

```
#ifndef __ADSPBLACKFIN__
    #warning   This program is written for Blackfin processors
#endif
```

C/C++ Preprocessor Features

Several features of the C/C++ preprocessor are used by CCES to control the programming environment. The `ccblkfn` compiler provides standard preprocessor functionality, as described in any C text. The following extensions to standard C are also supported:

- C++ Style Comments

- Preprocessor-Generated Warnings

- GCC Variable Argument Macros

This section contains:

- Predefined Macros

- Writing Preprocessor Macros

Predefined Macros

The `ccblkfn` compiler defines macros to provide information about the compiler, source file, and options specified. These macros can be tested, using `#ifdef` and related directives, to support your program's needs. Similar tailoring is done in the system header files.

NOTE: For the list of predefined assertions, see -A *name (tokens)* .

Macros, such as `__DATE__`, can be useful if incorporated into the text strings. The # operator within a macro body is useful in converting such symbols into text constructs.

The *Predefined Compiler Macros* table describes the predefined compiler macros.

Table 2-41: Predefined Compiler Macros

Macro	*Function*
_ADI_FX_LIBIO	Defined as 1 when compiling with the -fixed-point-io switch.
_ADI_COMPILER	Defined as 1.
_ADI_THREADS	Defined as 1 when compiling with the -threads switch.
__ADSPBF50x__	Defined as 1 when the target processor (set using the -proc switch) is the ADSP-BF504, ADSP-BF504F, or ADSP-BF506F processor.
__ADSPBF51x__	Defined as 1 when the target processor (set using the -proc switch) is the ADSP-BF512, ADSP-BF514, ADSP-BF516, or ADSP-BF518 processor.
__ADSPBF52x__	Defined as 1 when the target processor (set using the -proc switch) is the ADSP-BF522, ADSP-BF524, ADSP-BF526, ADSP-BF523, ADSP-BF525, or ADSP-BF527 processor.
__ADSPBF52xLP__	Defined as 1 when the target processor (set using the -proc switch) is the ADSP-BF522, ADSP-BF524, or ADSP-BF526 processor.
__ADSPBF53x__	Defined as 1 when the target processor (set using the -proc switch) is the ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF534, ADSP-BF536, ADSP-BF537, ADSP-BF538, or ADSP-BF539 processor.
__ADSPBF54x__	Defined as 1 when the target processor (set using the -proc switch) is the ADSP-BF542, ADSP-BF544, ADSP-BF547, ADSP-BF548, or ADSP-BF549 processor.
__ADSPBF56x__	Defined as 1 when the target processor (set using the -proc switch) is the ADSP-BF561 processor.
__ADSPBF59x__	Defined as 1 when the target processor (set using the -proc switch) is the ADSP-BF592-A processor.
__ADSPBF5xx__	Defined to 1 when building for any of the ADSP-BF5xx parts, equivalent to: `( defined(__ADSPBF50x__) || defined(__ADSPBF51x__) || defined(__ADSPBF52x__) || defined(__ADSPBF53x__) || defined(__ADSPBF54x__) || defined(__ADSPBF56x__) || defined(__ADSPBF59x__) )`
__ADSPBF60x__	Defined as 1 when the target processor (set using the -proc switch) is the ADSP-BF606, ADSP-BF607, ADSP-BF608 or ADSP-BF609 processor.
__ADSPBF6xx__	Equivalent to __ADSPBF60x__.
__ADSPBF70x__	Defined as 1 when the target processor (set using the -proc switch) is the ADSP-BF700, ADSP-BF701, ADSP-BF702, ADSP-BF703, ADSP-BF704, ADSP-BF705, ADSP-BF706 or ADSP-BF707 processor.
__ADSPBF71x__	Defined as 1 when the target processor (set using the -proc switch) is the ADSP-BF715, ADSP-BF716, ADSP-BF718 or ADSP-BF719 processor.
__ADSPBF7xx__	Defined to 1 when building for any of the ADSP-BF7xx parts, equivalent to: `( defined(__ADSPBF70x__) || defined(__ADSPBF71x__) )`
__ADSPBLACKFIN__	Always defined as 1.

Table 2-41: Predefined Compiler Macros (Continued)

Macro	Function
`__ADSPLPBLACKFIN__`	Defined as 1 for parts other than those in the ADSP-BF7xx family. For the ADSP-BF7xx processors it is defined as 0x220.
`__ADSPBF506F_FAMILY__`	Equivalent to `__ADSPBF50x__`.
`__ADSPBF518_FAMILY__`	Equivalent to `__ADSPBF51x__`.
`__ADSPBF526_FAMILY__`	Equivalent to `__ADSPBF52xLP__`.
`__ADSPBF527_FAMILY__`	Defined as 1 when the target processor (set using the `-proc` switch) is the ADSP-BF523, ADSP-BF525, or ADSP-BF527 processor.
`__ADSPBF533_FAMILY__`	Defined as 1 when the target processor (set using the `-proc` switch) is the ADSP-BF531, ADSP-BF532, or ADSP-BF533 processor.
`__ADSPBF537_FAMILY__`	Defined as 1 when the target processor (set using the `-proc` switch) is the ADSP-BF534, ADSP-BF536 or ADSP-BF537 processor.
`__ADSPBF538_FAMILY__`	Defined as 1 when the target processor (set using the `-proc` switch) is the ADSP-BF538 or ADSP-BF539 processor.
`__ADSPBF548_FAMILY__`	Defined as 1 when the target processor (set using the `-proc` switch) is the ADSP-BF542, ADSP-BF544, ADSP-BF547, ADSP-BF548, or ADSP-BF549.
`__ADSPBF548M_FAMILY__`	Defined as 1 when the target processor (set using the `-proc` switch) is the ADSP-BF542M, ADSP-BF544M, ADSP-BF547M, ADSP-BF548M, or ADSP-BF549M.
`__ADSPBF609_FAMILY__`	Equivalent to `__ADSPBF60x__`.
`__ADSPBF707_FAMILY__`	Equivalent to `__ADSPBF70x__`.
`__ADSPBF716_FAMILY__`	Equivalent to `__ADSPBF71x__`.
`__ANALOG_EXTENSIONS__`	Defined as 1 unless MISRA-C is enabled.
`__BASE_FILE__`	The preprocessor expands this macro to a string constant which is the current source file being compiled as seen on the compiler command-line.
`__CCESVERSION__`	The preprocessor defines this macro to be an eight-digit hexadecimal representation of the CCES release, in the form `0xMMmmUUPP`, where:- `MM` is the major release number- `mm` is the minor release number- `UU` is the update number- `PP` is the patch release number. For example, CrossCore Embedded Studio 1.1.0.0 would define __CCESVERSION__ as `0x01010000`.
`__cplusplus`	Defined in C++ mode, with value depending on language variant: • `199711L` for ISO/IEC 14882:2003 (default) • `201103L` for ISO/IEC 14882:2011 (enabled with the -c++11 switch) • `1` for GNU C++ (enabled with the -g++ switch) Also defined to `1` for LDF preprocessing.
`__DATE__`	The preprocessor expands this macro into the preprocessing date as a string constant. The date string constant takes the form `mm dd yyyy` (ANSI standard).

Table 2-41: Predefined Compiler Macros (Continued)

Macro	Function
`__DOUBLES_ARE_FLOATS__`	Defined as 1 when the size of the `double` type is the same as the single-precision `float` type. When the compiler `-double-size-64` switch is used (-double-size-{32 | 64}), the macro is not defined.
`__ECC__`	Always defined as 1.
`__EDG__`	Always defined as 1. This definition signifies that an Edison Design Group compiler front-end is being used.
`__EDG_VERSION__`	Always as an integral value representing the version of the compiler's front-end.
`__EXCEPTIONS`	Defined as 1 when C++ exception handling is enabled (using the -eh switch).
`__FILE__`	The preprocessor expands this macro into the current input file name as a string constant. The string matches the name of the file specified on the command line or in a preprocessor `#include` command (ANSI standard).
`__FIXED_POINT_ALLOWED`	Defined as 1 unless MISRA-C is enabled. It is defined to indicate that the native fixed-point types support may be used. For more information, see Using Native Fixed-Point Types.
`__HAS_L1_PARITY_CHECK__`	Defined as 1 when building for the ADSP-BF60x family parts.
`_HEAP_DEBUG`	Defined as 1 when Heap Debugging support is enabled, otherwise it is undefined. For more information, see Heap Debugging in the Achieving Optimal Performance From C/C++ Source Code chapter.
`__IDENT__`	The preprocessor expands `__IDENT__` to a string normally set using `#ident`.
`_INSTRUMENTED_PROFILING`	Defined as 1 when instrumented profiling is enabled (using the -p switch).
`_LANGUAGE_C`	Always defined as 1.
`__LINE__`	The preprocessor expands this macro into the current input line number as a decimal integer constant (ANSI standard).
`_LONG_LONG`	Always defined as 1 when compiling C and C++ sources to indicate that 64-bit double word integer types are supported.
`_MISRA_RULES`	Defined as 1 when compiling in MISRA-C mode.
`__NUM_CORES__`	ccblkfn defines `__NUM_CORES__` to the number of cores on the target Blackfin part.
`_PGO_HW`	Defined as 1 when you compile with both the `-pguide` and `-prof-hw` command-line switches (-pguide and -prof-hw).
`__RTTI`	Defined as 1 when C++ run-time type information is enabled (using the -rtti switch).
`__SIGNED_CHARS__`	Defined as 1, unless you compile with the `-unsigned-char` command-line switch (-unsigned-char).
`__SILICON_REVISION__`	ccblkfn defines `__SILICON_REVISION__` to a hexadecimal constant corresponding to the target processor revision. For more information, see Using the -si-revision Switch.
`__STDC__`	Always defined as 1.
`__STDC_VERSION__`	ccblkfn defines `__STDC_VERSION__` as 199409L when compiling in C89 mode, and as 199901L when compiling in C99 mode.

Table 2-41: Predefined Compiler Macros (Continued)

Macro	Function
`__TIME__`	The preprocessor expands this macro into the preprocessing time as a string constant. The date string constant takes the form `hh:mm:ss` (ANSI standard).
`__VERSION__`	Defined as a string constant giving the version number of the compiler used to compile this module.
`__VERSIONNUM__`	Defined as a numeric variant of `__VERSION__` constructed from the version number of the compiler. Eight bits are used for each component in the version number, and the most significant byte of the value represents the most significant version component. For example, a compiler with version 7.1.0.0 defines `__VERSIONNUM__` as `0x07010000` and 7.1.1.10 would define `__VERSIONNUM__` to be `0x0701010A`.
`__WORKAROUNDS_ENABLED`	Defines this macro as 1 if any hardware workarounds are implemented by the compiler. This macro is set if the `-si-revision` switch (-si-revision *version*) has a value other than `none` or if any specific workaround is selected by means of the `-workaround` switch (-workaround *workaround_id[, workaround_id]*).

Writing Preprocessor Macros

A *macro* is a user-defined name or string for which the preprocessor substitutes a user-defined block of text. Use the `#define` preprocessor command to create a macro definition. When a macro definition has arguments, the block of text the preprocessor substitutes can vary with each new set of arguments.

Compound Macros

Whenever possible, use inline functions rather than compound macros. If compound macros are necessary, define such macros to allow invocation like function calls. This makes your source code easier to read and maintain. If you want your macro to extend over more than one line, you must escape the newlines with backslashes. If your macro contains a string literal and you are using the `-no-multiline` switch (-no-multiline), escape the newline twice, once for the macro and once for the string.

The following two code segments define two versions of the macro `SKIP_SPACES`.

```
/* SKIP_SPACES, regular macro */
#define SKIP_SPACES (p, limit) {        \
   char *lim = (limit);                 \
      while ((p) != lim) {              \
         if (*(p)++ != ' ') {          \
            (p)--;                      \
            break;                      \
         }                              \
      }                                 \
}

/* SKIP_SPACES, enclosed macro */
#define SKIP_SPACES (p, limit)   \
   do {                          \
      char *lim = (limit);       \
```

```
        while ((p) != lim) {        \
            if (*(p)++ != ' ') {     \
                (p)--;               \
                break;               \
            }                        \
        }                            \
    } while (0)
```

Enclosing the first definition within the `do {} while (0)` pair changes the macro from expanding to a compound statement to expanding to a single statement. With the macro expanding to a compound statement, you would sometimes need to omit the semicolon after the macro call in order to have a legal program. This leads to a need to remember whether a function or macro is being invoked for each call and whether the macro needs a trailing semicolon or not. With the `do {} while (0)` construct, you can treat the macro as a function and put the semicolon after it.

For example,

```
/* SKIP_SPACES, enclosed macro, ends without `;' */
if (*p != 0)
    SKIP_SPACES (p, lim);
else
```

This expands to:

```
if (*p != 0)
    do {
        ...
    } while (0);
else ...
```

Without the `do {} while (0)` construct, the expansion would be:

```
if (*p != 0)
{
    ...
};    /* Probably not intended syntax */
else
```

C/C++ Run-Time Model and Environment

This section describes the Blackfin processor C/C++ run-time model and run-time environment. The C/C++ run-time model, which applies to compiler-generated code, includes descriptions of layout of the stack, data access, and call/entry sequence. The C/C++ run-time environment includes the conventions that C/C++ routines must follow to run on Blackfin processors. Assembly routines linked to C/C++ routines must follow these conventions.

The run-time environment issues include the following topics.

- Registers

- Managing the Stack

- Function Call and Return

- Data Storage Formats

- Memory Section Usage

- Global Array Alignment

- Controlling System Heap Size and Placement

- Using Multiple Heaps

- Startup and Termination

Registers

The compiler makes use of the processor's registers in a variety of ways, as shown in the *Processor Register Categorization* table. Some registers fulfill more than role, depending on context.

This section contains:

- Dedicated Registers

- Preserved Registers

- Scratch Registers

- Stack Registers

- Event Stack Register

- Call-Expansion Register

- Parameter Registers

- Return Registers

- Aggregate Return Register

- Comparison Return Register

- Reservable Register

Table 2-42: Processor Register Categorization

Register	Categorization
R0-R1	Scratch register, parameter register, return registers
R2	Scratch register, parameter register
R3	Scratch register
R4-R7	Preserved registers
P0	Scratch register, aggregate return register

Table 2-42: Processor Register Categorization (Continued)

Register	Categorization
P1	Scratch register, call-expansion register (not Blackfin+)
P2	Scratch register
P3-P5	Preserved register
SP, FP	Stack registers, dedicated registers
USP	Stack register, event stack register
ASTAT	Scratch register
CC,	Scratch register, comparison return register
I0-I3, B0-B3, M0-M2	Scratch register
M3	Scratch register, reservable register
L0-L3	Dedicated register
LT0-LT1, LB0-LB1	Scratch register
LC0-LC1	Dedicated register
A0-A1	Scratch register

Dedicated Registers

The C/C++ run-time environment specifies a set of registers whose contents should not be changed except in specific defined circumstances. If these registers are changed, their values must be saved and restored. The dedicated register values must always be valid:

- On entry to any compiled function.

- On return to any compiled function.

- On exit from `asm` statements and interrupt handlers.

The dedicated registers are `SP`, `FP`, `L0-L3` and `LC0-LC1`.

- `SP` and `FP` are the stack pointer and the frame pointer registers, respectively.

- The `L0-L3` registers define the lengths of the DAG's circular buffers. The compiler uses the DAG registers, both in linear mode and in circular buffering mode. The compiler assumes that the Length registers are zero, both on entry to functions and on return from functions, and ensures this is the case when it generates calls or returns. Your application may modify the Length registers and use the circular buffers, but you must ensure that the Length registers are appropriately reset when calling compiled functions, or returning to compiled functions. Interrupt handlers must save and restore the Length registers, if using DAG registers.

- The `LC0-LC1` registers are the hardware loop counters. They are normally considered scratch registers, but when the `-zero-loop-counters` switch (-zero-loop-counters) is specified, the compiler ensure that these registers are reset to zero on return from every compiled function, in case overlays or other code-movement techniques are in use.

When generating code for a function marked as an event handler, the compiler will emit code to save the current value of dedicated registers, and to re-establish the expected values.

Preserved Registers

These registers are also known as *callee-preserved registers*, as it is the callee's responsibility to ensure that these registers have the same value upon function return as they did upon entry to the function, regardless of whether the registers changed value in the meantime.

The C/C++ run-time environment specifies a set of registers whose contents must be saved and restored. Your assembly function must save these registers during the function's prologue and restore the registers as part of the function's epilogue. The call-preserved registers must be saved and restored if they are modified within the assembly function; if a function does not change a particular register, it does not need to save and restore the register. Usually, the registers are `P3-P5` and `R4-R7`.

> **NOTE:** Functions may declare a non-standard partitioning of preserved/ scratch registers through mechanisms such as #pragma regs_clobbered_call *string* , which any calling function must respect.

Scratch Registers

Scratch registers are also known as *caller-preserved* registers, as it is the caller's responsibility to ensure that the value of these registers is preserved across function calls, if required.

The C/C++ run-time environment specifies a set of registers whose contents need not be saved and restored. Note that the contents of these registers are not preserved across function calls.

The *Scratch Registers* table lists the scratch registers, supplying notes when appropriate.

Table 2-43: Scratch Registers

Scratch Register	Notes
P0	Used as the aggregate return pointer
P1-P2	On pre-Blackfin+ processors, `P1` is the Call-expansion Register.
R0-R3	The first three words of the argument list are always passed in `R0`, `R1`, and `R2` if present (`R3` is not used for parameters).
LB0-LB1	
LC0-LC1	Unless `-zero-loop-counters` switch is in effect.
LT0-LT1	
ASTAT	Including `CC`
A0-A1	
I0-I3	
B0-B3	
M0-M3	Unless `M3` is reserved

NOTE: Functions may declare a non-standard partitioning of preserved/ scratch registers through mechanisms such as #pragma regs_clobbered_call *string* , which any calling function must respect.

Loop Counters, Overlays and DMA'd Code

The compiler does not ensure that the loop counter registers (LC0 and LC1) are zero on entry or exit from a function. This does not normally cause a problem because the exit point of a hardware loop is unique within the program, and the compiler ensures that the only path to the exit is through the corresponding loop setup instruction.

If overlays are being used, or if code is being DMA'd into faster memory for execution, this may no longer be the case. It is possible for an overlay or a DMA'd function to set up a loop that terminates at address A, and then for a different overlay or DMA'd function to have different code occupying address A at a later point in time. If a hardware loop is still active - LC0 or LC1 is non-zero-at the point when the instruction at address A is rea ched, then undefined behavior results as the hardware loop "jumps" back to the start of the loop.

Therefore, in such cases, it is necessary for the overlay manager or the DMA manager to reset loop counters to ensure no hardware loops remain active that might relate to the address range covered by the variant code. A convenient way to achieve this is to use the -zero-loop-counters switch (-zero-loop-counters).

Stack Registers

The C/C++ run-time environment reserves a set of registers that control the run-time stack. These registers may be modified for stack management in assembly functions, but must be saved and restored. Never modify the stack registers within compiled functions.

The stack registers are:

- SP, the stack pointer
- FP, the frame pointer
- USP, the user stack pointer

Event Stack Register

CCES applications by default execute in Supervisor Mode at interrupt level IVG15 for performance reasons, and therefore do not usually make use of USP, the User Stack Pointer. However, USP is used during entry to, and exit from, Exception and NMI events. As with any event, the handler function must save context, but if CPLBs are enabled, a CPLB Data Miss event could occur during entry or exit if the top of the stack is not covered by an active Data CPLB. With interrupts, this is not a problem, but with Exception and NMI events, the processor is already operating at too high a priority level, leading to a double-exception fault.

To avoid this issue, the run-time libraries make use of USP as a temporary register while setting SP to point to a dedicated storage. The previous value of USP is not stored-it is always discarded.

Call-Expansion Register

The compiler issues function calls using *call-relative* instructions, for performance reasons. When linking applications for pre-Blackfin+ processors, the linker may need to convert some of these instructions into *call-via-pointer* instructions, if the call-relative instruction does not have sufficient capacity to express the offset between the call site

and its destination. This expansion can be controlled by the -jcs2l (-jcs2l) and -no-jcs2l (-no-jcs2l) switches.

When performing this expansion, the linker makes use of the P1 register to load the address for the called function.

When building for Blackfin+ processors, which support direct call instructions that can reach the entire address range, the P1 register is not used for call expansion.

Parameter Registers

When calling a function, the first three words of parameter data are passed to the callee in registers R0-R2.

Return Registers

When a function returns a value back to its caller, if the returned value is 64 bits or smaller in size, the value is returned in the R0 register and, if necessary, in the R1 register.

Aggregate Return Register

When a function returns a value back to its caller, if the returned value is larger than 64 bits in size, the value is returned in space reserved on the stack. This stack space is allocated by the caller, and a pointer to the start of the space is passed to the callee by the caller in the P0 register.

Comparison Return Register

The compiler generates calls to internal support routines to perform floating-point comparisons. For performance reasons, these internal routines return their result status in the CC bit of the ASTAT registers, rather than the R0 register.

Reservable Register

The M3 register can be reserved using the -reserve switch (-reserve *register[, register]*). When this register is reserved, the compiler does not generate code that uses this register.

Managing the Stack

The C/C++ run-time environment uses the run-time stack to store automatic variables and return addresses. The stack is managed by a frame pointer (FP) and a stack pointer (SP) and grows downward in memory, moving from higher to lower addresses.

The stack pointer points to the address of the value on the top of the stack, i.e. it points to the most-recently pushed value.

The stack and frame pointers must always contain 4-byte-aligned values. On processors with Blackfin architecture, a misaligned stack pointer will cause a Misaligned Data Access Exception if an interrupt occurs; on processors with Blackfin+ architecture, a misaligned stack pointer will incur a significant performance penalty.

Whenever storing data on the stack, you must always decrement the stack pointer first, so that any data on the stack has an address that is equal to or higher than the current stack pointer value. Otherwise, data may be corrupted by interrupt handlers as they will save and restore context onto the top of the stack.

A *stack frame* is a section of the stack used to hold information about the current context of the C/C++ program. Information in the frame includes local variables, compiler temporaries, and parameters for the next function.

The frame pointer serves as a base for accessing memory in the stack frame. Routines refer to locals, temporaries, and parameters by their offset from the frame pointer.

The *Example Run-Time Stack* figure shows an example section of a run-time stack.
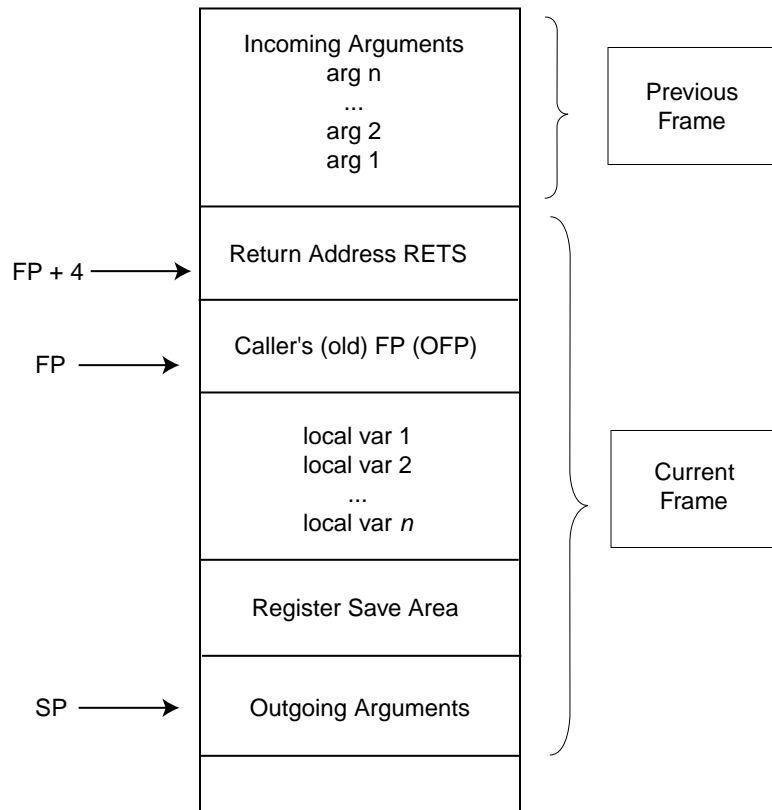
Figure 2-2: Example Run-Time Stack

In the *Example Run-Time Stack* figure, the currently executing routine, `Current()`, is called by `Previous()`, and `Current()` in turn calls `Next()`. The state of the stack is as if `Current()` has pushed all the arguments for `Next()` onto the stack and is just about to call `Next()`.

The compiler may omit using the frame pointer for "leaf" functions (functions which do not call other functions) for performance reasons, when optimizing.

Function Call and Return

The transfer of control from a calling function to a called function, and returning control back again, is the joint responsibility of the calling function and the called function. The calling function has to pass the appropriate parameters, in registers or upon the stack, and in some cases has to provide space for the return value too. The called

function has to keep its own temporary workspace separate from that of its caller. Both are required to ensure the integrity of some parts of the register set.

From the caller's point of view, the sequence of actions looks like this:

- Ensure that the return register, RETS, has been saved, as it will be clobbered by the call instruction. Normally, if a function is going to be calling any other functions, it saves RETS early on in its own prologue. In the *Example Run-Time Stack* figure, this is shown at FP+4.

- If the function being called clobbers registers that contain values being used by the caller, the caller must save those values on the stack prior to making the call. In the *Example Run-Time Stack* figure (see Managing the Stack), this is the "Register Save Area".

- If the called function returns an aggregate value that is returned via the stack, the caller must allocate stack space for this returned value. See Return Values.

- If the called function takes parameters, the caller must set up those parameters, either in registers or on the stack. In the *Example Run-Time Stack* figure, this is the "Outgoing Arguments".

- The caller can now call the function.

- After the function returns, the caller must reset the stack pointer, to dispose of the "Outgoing Arguments" space, and restore any needed registers that might have been clobbered by the called function.

From the callee's point of view, the sequence of actions looks like this:

- Upon entry to the callee, the stack pointer points to the top of the "Incoming Arguments" area of the *Example Run-Time Stack* figure. Note that this figure is viewed differently by caller and callee: the "Outgoing Arguments" of the caller are the "Incoming Arguments" as far as the callee is concerned.

- If the function is calling any further functions, it has to save the Frame Pointer and RETS, the Return Register. If it needs any space for temporaries, it must create the "local var" space on the stack. These operations are all combined by the LINK instruction.

- If the function needs to modify any registers that are not considered scratch registers, the function must save their current values prior to changing them. In other words, the function must preserve the value of any callee-preserved registers.

- The function may now perform its main task.

- Upon completion, the function may need to return a value to the caller. To do this, it must either load the value into the Result Registers, or store it to the stack.

- Prior to returning, the function must restore the value of any callee-preserved registers it has modified.

- The function must pop the "local var" space from the stack, restore the RETS value, restore the caller's Frame Pointer value (if changed) and restore the Stack Pointer to the value it had on entry to the function. These operations are all combined by the UNLINK instruction.

- Finally, the function can return control back to the caller.

Transferring Function Arguments and Return Value

The C/C++ run-time environment uses a set of registers and the run-time stack to transfer function parameters to assembly routines. Your assembly language functions must follow these conventions when they call (or when called by) C/C++ functions.

Basic Argument Passing

The basic details for argument passing are as follows:

- 8- and 16-bit arithmetic types must be sign- or zero-extended to 32 bits by the caller.

- 40-bit fixed-point types must be sign- or zero-extended to 64 bits by the caller.

- Parameters are pushed onto the stack in reverse order, with each parameter beginning on a 32-bit boundary. Thus, for a function that takes five `int` parameters a, b, c, d and e, the parameters' respective stack positions would be `SP`, `SP+4`, `SP+8`, `SP+12` and `SP+16`.

- However, although stack space is allocated for *all* parameters, the first twelve bytes are passed in the registers `R0-R2`. The first 32 bits are passed in `R0`, the second 32 bits in `R1` and the third 32 bits in `R2`. Thus, given the same five int parameters, parameter a would be passed in `R0`, parameter b would be passed in `R1`, and parameter c would be passed in R2.

   NOTE: When calling a C function, at least twelve bytes of stack space must be allocated for the function's arguments, corresponding to `R0-R2`. This applies even for functions with fewer than 12 bytes of argument data, or that have fewer than three arguments. Note that the called function is permitted to modify the contents of this stack space.

Passing Parameters for Variable Argument Lists

The details of argument passing do not change for variable argument lists.

For example, a function declared as follows may receive one or more arguments.

```
int varying(char *fmt, ...) { /* ... */ }
```

As with other functions, the first argument, `fmt`, is passed in `R0`, and other arguments are passed in `R1`, and then `R2`, and then on the stack, as required.

Variable argument lists are processed using the macros defined in the `stdarg.h` header file. The `va_start()` function obtains a pointer to the list of arguments which may be passed to other functions, or which may be walked by the `va_arg()` macro.

To support this, the compiler begins variable argument functions by flushing `R0`, `R1`, and `R2` to their reserved spaces on the stack:

```
_varying:
   [SP+0] = R0;
   [SP+4] = R1;
   [SP+8] = R2;
```

The `va_start()` function can then take the address of the last non-varying argument (`fmt`, in the example above, at `[SP+0]`), and `va_arg()` can walk through the complete argument list on the stack.

Passing a C++ Class Instance

A C++ class instance function parameter is always passed by reference when a copy constructor has been defined for the C++ class. If a copy constructor has not been defined for the C++ class then the C++ class instance function parameter is passed by value.

Consider the following example.

```
class fr
{
   public:
       int v;
   public:
       fr () {}
       fr (const fr& rc1) : v(rc1.v) {}
};

extern int fn(fr x);
fr Y;
int main() {
    return fn (Y);
}
```

The function call `fn (Y)` in `main` passes the C++ class instance `Y` by reference because a copy constructor for that C++ class has been defined by `fr (const fr& rc1) : v(rc1.v) {}`. If this copy constructor were removed, then `Y` would be passed by value.

Return Values

Values are usually returned from a called function to the caller in register `R0`, or in the register pair `R0-R1`, if necessary. The details are as follows:

- 8- and 16-bit arithmetic values are returned in `R0`, sign- or zero-extended to 32 bits as required.

- 32-bit arithmetic values are returned in `R0`.

- 40-bit fixed-point types are sign- or zero-extended to 64 bits, and returned in `R0` and `R1`, with the least significant bits in `R0`.

- 64-bit arithmetic types are returned in `R0` and `R1`, with the least significant bits in `R0`.

- Pointer values are returned in `R0`.

- Aggregate types of 32 bits or less are returned in `R0`.

- Aggregate types larger than 32 bits but less than or equal to 64 bits in size are returned in `R0` and `R1`, with the lower-addressed bytes in `R0`.

- Aggregate values larger than 64 bits in size are returned on the stack. The caller must allocate sufficient space on the stack within the caller's own frame, and load the address of the lowest-addressed part of this storage into register `P0` before calling the function.

Parameter and Return Value Examples

The *Examples of Parameter Passing* table provides examples of passed parameters.

Table 2-44: Examples of Parameter Passing

Function Prototype	Parameters Passed as	Return Location
`int test(int a, int b, int c)`	a in R0, b in R1, c in R2	in R0
`char test(int a, char b, char c)`	a in R0, b in R1, c in R2	in R0
`int test(int a)`	a in R0	in R0

`int test(char a,`
`        char b,`
`        char c,`
`        char d,`
`        char e)`
	a in R0,
	b in R1,
	c in R2,
	d in [FP+20],
	e in [FP+24]
		in R0

`int test(struct *a, int b, int c)`	a (addr) in R0, b in R1, c in R2	in R0

`struct s2a {`
`    char ta;`
`    char ub;`
`    int vc;}`
`int test(struct s2a x,  int b, int c)`
	x.ta and x.ub in R0,
	x.vc in R1,
	b in R2,
	c in [FP+20]
		in R0

`struct foo *test(int a,`
`              int b,`
`              int c)`
	a in R0,
	b in R1,
	c in R2
		(address) in R0

`void qsort(`
`    void *base,`
`    int  nel,`
`    int  width,`
`    int (*compare)(const void *,`
`            const void *))`
	base(addr) in R0,
	nel in R1,
	width in R2,
	compare(addr) in [FP+20]	

Table 2-44: Examples of Parameter Passing (Continued)

Function Prototype	Parameters Passed as	Return Location
```struct s2 { char t; char u; int v; } struct s2 test(int a, int b, int c)```	a in R0, b in R1, c in R2	in R0 (s.t and s.u) and in R1 (s.v)
```struct s3 { char t; char u; int v; int w; } struct s3 test(int a, int b, int c)```	a in R0, b in R1, c in R2	in *P0 (based on value of P0 at the call, not necessarily at the return)

Calling Assembly Subroutines From C/C++ Programs

Before calling an assembly language subroutine from a C/C++ program, create a prototype to define the arguments for the assembly language subroutine and the interface from the C/C++ program to the assembly language subroutine. Even though it is legal to use a function without a prototype in C/C++, prototypes are a strongly-recommended practice for good software engineering. When the prototype is omitted, the compiler cannot perform argument-type checking and assumes that the return value is of type integer and uses K&R promotion rules instead of ANSI promotion rules.

C, C++ and assembly code use different namespaces for symbols. Refer to Symbol Names in C/C++ and Assembly for ways to specify an assembly routine from C/C++.

The compiler assumes that the called assembly function obeys the run-time model's rules on register usage. Refer to Registers for details.

> **NOTE:** Functions may declare a non-standard partitioning of preserved/ scratch registers through mechanisms such as #pragma regs_clobbered *string*, which any calling function must respect. If the assembly function being called from C/C++ uses a non-standard clobber set, declare this in the prototype.

The compiler also assumes the machine state does not change during execution of the assembly language subroutine. If you change modes within your assembly routine, for example, the rounding-mode bit RND_MOD- ensure that you restore them to their previous value before returning.

Calling C/C++ Functions From Assembly Programs

C/C++ functions can be called from assembly code. The situation is similar to that described in Calling Assembly Subroutines From C/C++ Programs.

- The namespaces for C/C++ and assembly code are different; refer to Symbol Names in C/C++ and Assembly for details on how to specify a C/C++ function that can be referenced from assembly.

- The C/C++ function will obey the run-time model's rules described in Registers, so your calling assembly code must respect this, by not expecting caller-preserved registers to maintain their values over the call.

- If your assembly code is passing parameters to the C/C++ function or receiving a return value from it, you must follow the rules described in Transferring Function Arguments and Return Value.

There are additional requirements you must fulfill when calling C/C++ code from assembly code, however:

- You must ensure that the system stack is valid and appropriately aligned, as described in Managing the Stack.

- You must ensure that Dedicated Registers have their correct values.

- You must ensure that a system heap is set up. This is done for you if you are using the default or generated startup code and `.ldf` files. For more information, see Startup and Termination.

Symbol Names in C/C++ and Assembly

You can use C/C++ symbols (function or variable names) in assembly routines and use assembly symbols in C/C++ code. This section describes how to name and use C/C++ and assembly symbols.

Only global C/C++ symbols can be referenced from assembly source.

To use a C/C++ function or variable in an assembly routine, declare it as global in the C program. Import the symbol into the assembly routine by declaring the symbol with the `.EXTERN` assembler directive.

To use an assembly function or variable in your C/C++ program, declare the symbol with the `.GLOBAL` assembler directive in the assembly routine and import the symbol by declaring the symbol as `extern` in the C program.

The ***C/C++ Naming Conventions for Symbols*** table shows several examples of the C/C++ and assembly interface naming conventions.

Table 2-45: C/C++ Naming Conventions for Symbols

In C/C++ Program	*In Assembly Subroutine*
`int c_var; /*declared global*/`	`.extern _c_var;` `.type _c_var,STT_OBJECT;`
`void c_func(void);`	`.global _c_func;` `.type _c_func,STT_FUNC;`

`extern int asm_var;`	`.global _asm_var;`
	`.type _asm_var,STT_OBJECT;`
	`.byte = 0x00,0x00,0x00,0x00`

`extern void asm_func(void);`	`.global _asm_func;`
	`.type _asm_func,STT_FUNC;`
	`_asm_func:`

C/C++ and Assembly: Extern Linkage

The compiler supports the use of `extern` to declare symbol names in the different C, C++ and assembly namespaces. For example,

```
extern int def_fn(void);      // "_def_fn" or "__Z6def_fnv"
extern "asm" int asm_fn(void); // "asm_fn" in assembly
extern "C" int c_fn(void);     // "_c_fn" in assembly
```

When compiling your source in C or C++ mode, you can use `extern "asm"` or `extern "C"` to specify which namespace you want your external symbols to use. Without the external linkage specifier, your symbol will use C namespace when compiling in C mode, and C++ namespace (mangled) when compiling in C++ mode.

C and Assembly: Underscore Prefix

As can be seen in C/C++ and Assembly: Extern Linkage, when the compiler generates the assembly version of a C-namespace symbol, it prepends an underscore. You can take advantage of this in your assembly source when refer-ring to C-mode symbols, by adding the underscore yourself.

Other Approaches

In addition to the external linkage feature described in C/C++ and Assembly: Extern Linkage, you can also use the following approaches in your C/C++ source:

- When declaring functions, provide an alternative linkage name using #pragma linkage_name *identifier* .

- When declaring variables in C, provide an alternative linkage name using (asm) Keyword for Specifying Names in Generated Assembler.

- When declaring functions in C, use Function, Variable, and Type Attribute Keyword (__attribute__) to specify aliases of functions.

Exceptions Tables in Assembly Routines

C++ functions can throw C++ exceptions, which must be caught by another function earlier in the call-stack. Part of this catching process involves unwinding the stack of intervening, still-active function calls. The C++ exception sup-port library uses additional function details to perform this unwinding. The exception support gets this information from different places:

- When C++ modules are compiled with exceptions enabled by the `-eh` switch (-eh), the compiler generates the necessary unwinding tables.

- When C modules are compiled, exceptions information is not usually necessary, but the compiler will generate unwinding information if the `-eh` switch is specified.

- Assembly modules are not compiled, so unwinding information must be supplied manually, if necessary.

Assembly functions rarely need to provide exception-unwinding information. It is only necessary when all of the following conditions apply:

- The assembly routine may be called by a C or C++ function.

- The assembly routine calls a C++ function (or a C function that may lead to a C++ function being called, while the assembly routine is still active).

- The called C++ function may throw an exception.

The assembly routine must allocate a stack frame using `FP` and `SP` as described in Managing the Stack. On entry to the assembly routine, call-preserved registers (Preserved Registers) that are modified in the routine should be saved into a contiguous region within the stack frame, called the *save* area. Registers are saved at ascending addresses in the save area in the order given in the ***Function Exception Table Register Numbers*** table.

A word in the `.gdt` section must be initialized with the address of the function exceptions table. This word must be marked with the `.RETAIN_NAME` directive to prevent it being removed by linker data elimination. The function exceptions table itself must be initialized as illustrated in the *Function Exceptions* table.

Table 2-46: Function Exceptions

Offset	Size in Bytes	Meaning
0	4	Start address of the routine
4	4	First address after end of routine
8	4	Signed offset from FP of register save area
12	8	Bit set indicating which registers are saved
20	4	Always zero. Indicates this is not C++ code

The bit set field of the function exceptions table contains a bit for each register. The bits corresponding to registers saved in the save area must be set to one and the other bits set to zero. The bit numbers corresponding to each register are given in the *Function Exception Table Register Numbers* table, where bit 0 is the least significant bit of the lowest addressed word, bit 31 is the most significant bit of that word, bit 32 is the least significant bit of the second lowest addressed word, and so on.

Bit numbering may best be explained by the C code to test bit number.

```
int wrd = r/32; int bit = lu (r%32);
if (bitset[wrd] & bit) /* register r was saved */
```

Table 2-47: Function Exception Table Register Numbers

Register	Bit Number	Bytes Taken in Save Area if Saved
LB1	0	4
LB0	1	4
LT1	2	4
LT0	3	4
LC1	4	4
LC0	5	4
M3	6	4
M2	7	4
M1	8	4
M0	9	4
B3	10	4
B2	11	4
B1	12	4

Table 2-47: Function Exception Table Register Numbers (Continued)

Register	Bit Number	Bytes Taken in Save Area if Saved
B0	13	4
I3	14	4
I2	15	4
I1	16	4
I0	17	4
L3	18	4
L2	19	4
L1	20	4
L0	21	4
A1X	22	4
A1W	23	4
A0X	24	4
A0W	25	4
P5	26	4
P4	27	4
P3	28	4
P2	29	4
P1	30	4
P0	31	4
R7	32	4
R6	33	4
R5	34	4
R4	35	4
R3	36	4
R2	37	4
R1	38	4
R0	39	4
ASTAT	40	4

This example shows an assembly routine with function exceptions table,

```
.section program;
    _asmfunc: .LN._asmfunc:        LINK 0;
    /* setup FP */
    [--SP] = (R7:5, P5:4);
```

```
        /* save R5,R6,R7,P4,P5 at FP-20 */
        /* use R5,R6,R7,P4,P5 call a C++ function */
        (R7:5, P5:4) = [SP++];
        /* restore registers */
        UNLINK;
        RTS;
.LN._asmfunc.end: ._asmfunc.end:
        .global _asmfunc;
        .type _asmfunc, STT_FUNC;
        .section .edt;
        /* conventionally function exceptions tables go in .edt */
        .align 4;
        .byte4 .function_exceptions_table[6] =
        .LN._asmfunc,              /* first address of _asmfunc */
        .LN._asmfunc.end,          /* first address after _asmfunc */
        -20,                       /* offset of save area from FP */
        0x0c000000, 0x00000007,    /* bit set, bits 26=P5,
                                      27=P4,32=R7,33=R6,34=R5 */
        0;                         /* always zero for non-c++ */
        .section .gdt;
        .align 4;
        .fet_index:    .byte4 = .function_exceptions_table;
                                   /* address of table in .gdt */
        .retain_name .fet_index;
```

Data Storage Formats

The sizes of intrinsic C/C++ data types are selected by Analog Devices so that normal C/C++ programs execute with hardware-native data types, and, therefore, at high speed. The C/C++ run-time environment uses the intrinsic C/C++ data types and data formats that appear in the *Data Storage Formats and Data Type Sizes* table and are shown in the *Data Storage Format for Float and Double Types* figure and the *Double-Precision IEEE Format* figure.

Table 2-48: Data Storage Formats and Data Type Sizes

Type	Bit Size	Number Representation	sizeof Returns
bool	8 bits signed	8-bit two's complement	1
char	8 bits signed	8-bit two's complement	1
unsigned char	8 bits unsigned	8-bit unsigned magnitude	1
short	16 bits signed	16-bit two's complement	2
unsigned short	16 bits unsigned	16-bit unsigned magnitude	2
int	32 bits signed	32-bit two's complement	4
unsigned int	32 bits unsigned	32-bit unsigned magnitude	4
long	32 bits signed	32-bit two's complement	4
unsigned long	32 bits unsigned	32-bit unsigned magnitude	4

Table 2-48: Data Storage Formats and Data Type Sizes (Continued)

Type	Bit Size	Number Representation	sizeof Returns
long long	64 bits signed	64-bit two's complement	8
unsigned long long	64 bits unsigned	64-bit unsigned magnitude	8
pointer	32 bits	32-bit two's complement	4
function pointer	32 bits	32-bit two's complement	4
double	32 bits	32-bit IEEE single-precision	4
float	32 bits	32-bit IEEE single-precision	4
double	64 bits	64-bit IEEE double-precision	8
long double	64 bits	64-bit IEEE	8
short fract	16 bits signed	s1.15 fract	2
fract	16 bits signed	s1.15 fract	2
long fract	32 bits signed	s1.31 fract	4
unsigned short fract	16 bits unsigned	0.16 fract	2
unsigned fract	16 bits unsigned	0.16 fract	2
unsigned long fract	32 bits unsigned	0.32 fract	4
short accum	40 bits signed	s9.31 fixed-point	8
accum	40 bits signed	s9.31 fixed-point	8
long accum	40 bits signed	s9.31 fixed-point	8
unsigned short accum	40 bits unsigned	8.32 fixed-point	8
unsigned accum	40 bits unsigned	8.32 fixed-point	8
unsigned long accum	40 bits unsigned	8.32 fixed-point	8
fract16	16 bits signed	1.15 fract	2
fract32	32 bits signed	1.31 fract	4

NOTE: The floating-point and 64-bit data types are implemented using software emulation, and are expected to run more slowly than hardware-supported native data types. The emulated data types are `float`, `double`, `long double`, `long long`, and `unsigned long long`.

CAUTION: The native fixed-point types `fract` and `accum` are available only when the `stdfix.h` header file is included.

CAUTION: The `fract16` and `fract32` are not actually intrinsic data types-they are `typedefs` to `short` and `long`, respectively. You need to use built-in functions to do basic arithmetic on these types. (See Fractional Value Built-In Functions.) You cannot do `fract16*fract16` and get the right result. The native fixed-point types `fract` and `accum` provide a more natural alternative to `fract16` and `fract32`.

Floating-Point Data Size

On Blackfin processors, the `float` data type is 32 bits, and the `double` data type default size is 32 bits. This size is chosen because it is the most efficient. The 64-bit `long double` data type is available if more precision is needed, although this is more costly because the type exceeds the data sizes supported natively by hardware.

In the C language, floating-point literal constants default to the `double` data type. When operations involve both `float` and `double`, the `float` operands are promoted to `double` and the operation is done at `double` size. By having `double` default to a 32-bit data type, the Blackfin compiler usually avoids additional expense during these promotions. This does not, however, fully conform to the ISO/IEC 9899:1990 C standard, the ISO/IEC 9899:1999 C standard, and the ISO/IEC 14882:2003 C++ standard, all of which require that the `double` type supports at least 10 digits of precision.

The `-double-size-64` switch (-double-size-{32 | 64}) sets the size of the `double` type to 64 bits if additional precision, or full standard conformance, is required.

The `-double-size-64` switch causes the compiler to treat the `double` data type as a 64-bit data type, instead of a 32-bit data type. This means that all values are promoted to 64 bits, and consequently incur more storage and cycles during computation. The switch does not affect the size of the `float` data type, which remains at 32 bits.

Consider the following case.

```
float add_two(float x) { return x + 2.0; } // has promotion
```

When compiling this function, the compiler promotes the `float` value x to `double`, to match the literal constant `2.0`. The addition becomes a `double` operation, and the result is truncated back to a `float` before being returned.

By default, or with the `-double-size-32` switch (-double-size-{32 | 64}), the promotion and truncation operations are empty operations-they require no work because the `float` and `double` types default to the same size. Thus, there is no cost.

With the `-double-size-64` switch, the promotion and truncation operations require work because the `double` constant 2.0 is a 64-bit value. The x value is promoted to 64 bits, a 64-bit addition is performed, and the result is truncated to 32 bits before being returned.

In contrast, since the literal constant `2.0f` in the following example has an "f" suffix, it is a float-type constant, not a double-type constant.

```
float add_two(float x) { return x + 2.0f; } // no promotion
```

Thus, both operands to the addition are of type `float`, and no promotion or truncation is necessary. This version of the function does not produce any performance degradation when the `-double-size-64` switch is used.

You must be consistent in your use of the `-double-size-{32|64}` switch.

Consider the two files, such as:

```
file x.c:
  double add_nums(double x, double y) { return x + y; }
file y.c:
```

```
extern double add_nums(double, double);
double times_two(double val) { return add_nums(val, val); }
```

Both files must be compiled with the same usage of -double-size{32|64}. Otherwise, times_two() and add_nums() will be exchanging data in mismatched formats, and incorrect behavior will occur. The *-double-size-{32|64} Switch* table shows the results for the various permutations.

Table 2-49: -double-size-{32|64} Switch

x.c	*y.c*	*Result*
default	default	Okay
default	-double-size-32	Okay
-double-size-32	default	Okay
-double-size-32	-double-size-32	Okay
-double-size-64	-double-size-64	Okay
-double-size-32	-double-size-64	Error
-double-size-64	-double-size-32	Error

If a file does not make use of any double-typed data, it may be compiled with the -double-size-any switch, to indicate this fact. Files compiled in this way may be linked with files compiled with -double-size-32 or with -double-size-64, without conflict.

Conflicts are detected by the linker and result in linker error li1151, *"Input sections have inconsistent qualifiers"*.

Floating-Point Binary Formats

The Blackfin compiler supports IEEE floating-point format.

IEEE Floating-Point Format

By default, the Blackfin compiler provides floating-point emulation using IEEE single- and double-precision formats. Single-precision IEEE format (*Data Storage Format for Float and Double Types* figure) provides a 32-bit value, with 23 bits for the mantissa, 8 bits for the exponent, and 1 bit for the sign. This format is used for the float data type, and for the double data type by default and when the -double-size-32 switch is used. The 32-bit double data type violates the ISO/IEC 9899:1990 C standard, the ISO/IEC 9899:1999 C standard, and the ISO/IEC 14882:2003 C++ standard.
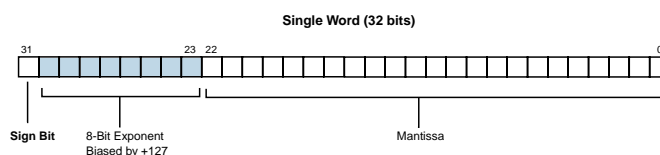
Figure 2-3: Data Storage Format for Float and Double Types

In the *Data Storage Format for Float and Double Types* figure, the single word (32-bit) data storage format equates to:

$-1^{Sign} \times 1.\text{Mantissa} \times 2^{(Exponent - 127)}$

where:

- Sign - Comes from the sign bit.

- Mantissa - Represents the fractional part of the mantissa, 23 bits. (The "1." is assumed in this format.)

- Exponent - Represents the 8-bit exponent.

Double-precision IEEE format (the *Double-Precision IEEE Format* figure) provides a 64-bit value, with 52 bits for the mantissa, 11 bits for the exponent, and 1 bit for the sign. This format is used for the `long double` data type, and for the `double` data type when the `-double-size-64` switch is used. A 64-bit value for the `double` data type is compliant to with the ISO/IEC 9899:1990 C standard, the ISO/IEC 9899:1999 C standard, and the ISO/IEC 14882:2003 C++ standard. (See Language Standards Compliance.)
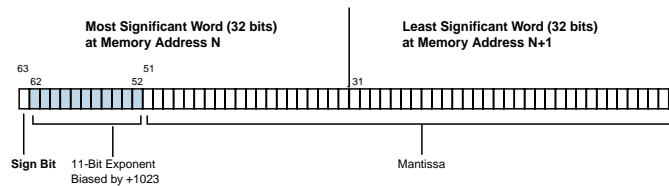
Figure 2-4: Double-Precision IEEE Format

In the *Double-Precision IEEE Format* figure, the two-word (64-bit) data storage format equates to:

$-1^{Sign} \times 1.\text{Mantissa} \times 2^{(Exponent - 1023)}$

where:

- Sign - Comes from the sign bit

- Mantissa - Represents the fractional part of the mantissa, 52 bits. The "1." is assumed in this format.

- Exponent - Represents the 11-bit exponent

IEEE Floating-Point Implementation

The Blackfin compiler supports a high-performance implementation of IEEE floating-point, which relaxes some of the IEEE rules in the interest of performance:

- The Round-To-Nearest-Even mode is the only supported rounding mode.

- Exception flags are not supported.

- There is no distinction between signaling NaN and a quiet NaN; all NaNs are handled as quiet NaNs.

- In general, de-normalized numbers are flushed to zero before being used in arithmetic.

- The emulation routines do not stringently observe all of the rules of the IEEE standard with respect to the handling of Infinity or NaN; for example, a division by Infinity always returns a NaN, even when the numerator is not zero.

- When a floating-point operation generates a result of zero, the emulation routines do not always ensure that it has the correct sign. For example, when a double precision value that is less than `FLT_MIN` in magnitude is converted to single precision, the sign of the result will always be +0.0 irrespective of the sign of the double precision value.

- Double precision arithmetic may occasionally not round correctly and lose one bit of precision.

fract and accum Data Representation

The `fract` and `accum` types are native fixed-point types that can be used to write code using saturating, fixed-point arithmetic. They should not be confused with the `fract16` and `fract32` typedefs which may be used to write fixed-point arithmetic via built-in functions only. The native fixed-point types are discussed in Using Native Fixed-Point Types.

The `short fract` and `fract` types represent a single 16-bit signed fractional value, while the `long fract` type represents a 32-bit signed fractional value. Both types have the same range, [-1.0,+1.0). However, `long fract` has twice the precision.

The `short fract`, `fract`, and `long fract` data representations are shown in the *Data Storage Format for short fract, fract, and long fract* figure.

Short fract, fract (1.15)

Bit	15	14	13		2	1	0
Weight	(-1)	2^{-1}	2^{-2}		2^{-13}	2^{-14}	2^{-15}

Long fract (1.31)

Bit	31	30	29		2	1	0
Weight	(-1)	2^{-1}	2^{-2}		2^{-29}	2^{-30}	2^{-31}
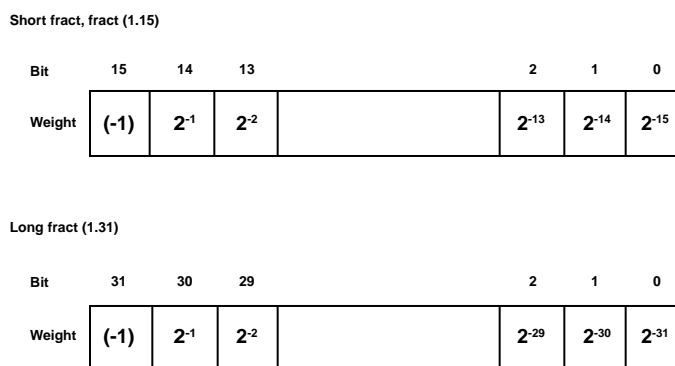
Figure 2-5: Data Storage Format for `short fract`, `fract`, and `long fract`

Therefore, to represent 0.25 in `fract`, the HEX representation would be `0x2000` (2^{-2}). For -0.25 in `long fract`, the HEX representation is `0xe000 0000` ($-1+2^{-1}+2^{-2}$). For -1, the HEX representation in `fract` is `0x8000`. `short fract`, `fract`, and `long fract` cannot represent +1 exactly, but they get quite close with `0x7fff` for `short fract` and `fract`, or `0x7fff ffff` for `long fract`.

The `unsigned short fract` and `unsigned fract` types represent a single 16-bit unsigned fractional value, while the `unsigned long fract` type represents a 32-bit unsigned fractional value. Both types have the same range, [0.0,+1.0). However, `unsigned long fract` has twice the precision.

The `unsigned short fract,` `unsigned fract` and `unsigned long fract` data representations are shown in the ***Data Storage Format for unsigned short fract, unsigned fract, and unsigned long fract*** figure.

Unsigned short fract, unsigned fract (0.16)

Bit	15	14	13		2	1	0
Weight	2^{-1}	2^{-2}	2^{-3}		2^{-14}	2^{-15}	2^{-16}

Unsigned long, unsigned fract (0.32)

Bit	31	30	29		2	1	0
Weight	2^{-1}	2^{-2}	2^{-3}		2^{-30}	2^{-31}	2^{-32}
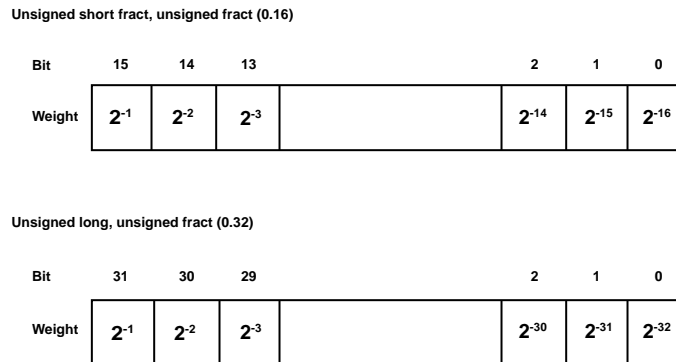
Figure 2-6: Data Storage Format for `unsigned short fract,` `unsigned fract,` and `unsigned long fract`

Therefore, to represent 0.25 in `unsigned fract`, the HEX representation would be `0x4000` (2^{-2}). For 0.125 in `unsigned long fract`, the HEX is `0x2000 0000` (2^{-3}). `unsigned short fract,` `unsigned fract` and `unsigned long fract` cannot represent +1 exactly, but they get quite close with `0xffff` for `unsigned short fract` and `unsigned fract`, or `0xffff ffff` for `unsigned long fract`.

The `short accum,` `accum,` and `long accum` types represent a single 40-bit signed fixed-point value. The three types have the same range, [-256.0,+256.0). They should not be confused with the `acc40` type, which is a container for a value held in the accumulator register.

The `short accum,` `accum,` and `long accum` data representations are shown in the ***Data Storage Format for short accum, accum, and long accum*** figure.

short accum, accum, long accum (9.31)

Bit	39	38	37		2	1	0
Weight	(-2^8)	2^7	2^6		2^{-29}	2^{-30}	2^{-31}
	Sign Bit						
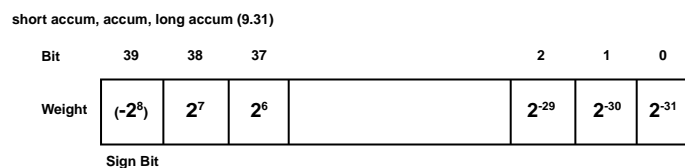
Figure 2-7: Data Storage Format for `short accum,` `accum,` and `long accum`

Therefore, to represent 12.25 in any of the signed accum types, the HEX representation would be `0x06 2000 0000` ($2^3+2^2+2^{-2}$). For -256.0, the HEX representation in the signed accum types is `0x80 0000 0000.` `short accum,` `accum,` and `long accum` cannot represent +256.0 exactly, but they get quite close with `0x7f ffff ffff`.

The `unsigned short accum` and `unsigned accum` types represent a single 40-bit unsigned fixed-point value. The three types have the same range, [0.0,+256.0).

The `unsigned short accum,` `unsigned accum,` and `unsigned long accum` data representations are shown in the ***Data Storage Format for unsigned short accum, unsigned accum, and unsigned long accum*** figure.

Unsigned short accum, unsigned accum, unsigned long accum (8.32)

Bit	39	38	37		2	1	0
Weight	2^7	2^6	2^5		2^{-30}	2^{-31}	2^{-32}
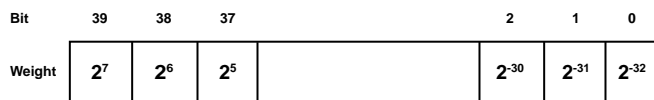
Figure 2-8: Data Storage Format for `unsigned short accum`, `unsigned accum`, and `unsigned long accum`

Therefore, to represent 12.25 in any of the unsigned accum types, the HEX representation would be `0x0c 4000 0000` ($2^3+2^2+2^{-2}$). `unsigned short accum`, `unsigned accum`, and `unsigned long accum` cannot represent +256.0 exactly, but they get quite close with `0xff ffff ffff`.

fract16 and fract32 Data Representation

The `fract16` type represents a single 16-bit signed fractional value, and the `fract32` type represents a 32-bit signed fractional value. Both types have the same range, [-1.0,+1.0). However, `fract32` has twice the precision. They are not intrinsic data storage formats, they are simply `typedefs`. They should therefore not be confused with the native fixed-point types, `fract` and `accum`, defined in the `stdfix.h` header file.

```
typedef short fract16;
```

typedef long fract32;

The `fract` data representation is shown in *Data Storage Format for fract16 and fract32* figure.

Signed Fractional (1.15)

Bit	15	14	13		2	1	0
Weight	(-1)	2^{-1}	2^{-2}		2^{-14}	2^{-14}	2^{-15}

Sign Bit

Signed Fractional (1.31)

Bit	31	30	29		2	1	0
Weight	(-1)	2^{-1}	2^{-2}		2^{-29}	2^{-30}	2^{-31}

Sign Bit
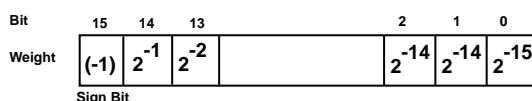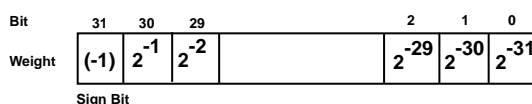
Figure 2-9: Data Storage Format for fract16 and fract32

Therefore, to represent 0.25 in `fract16`, the HEX representation would be `0x2000` (2^{-2}). For -0.25 in `fract32`, the HEX would be `0xe000 0000` ($-1+2^{-1}+2^{-2}$). For -1, the HEX representation in `fract16` would be `0x8000` (-1). `fract16` and `fract32` cannot represent +1 exactly, but they get quite close with `0x7fff` for `fract16`, or `0x7fff ffff` for `fract32`. There is also a `fract2x16` data type, which is two `fract16s` packed into 32 bits. The first two bytes belong to one `fract16`, and the second two bytes belong to the other. There are also built-in functions that work with `fract2x16` parameters.

Memory Section Usage

The C/C++ run-time environment requires that a specific set of memory section names are used to place code in memory. In assembly language files, these names are used as labels for the `.SECTION` directive. In the `.ldf` file,

these names are used as labels for the output section names within the SECTIONS{} command. For information on .ldf file syntax and other information on the linker, see the *Linker and Utilities Manual.*

Code Storage

The code section, program, is where the compiler puts all the program instructions that it generates when compiling the program. The cplb_code section exists so that memory protection management routines can be placed into sections of memory that are always configured as being available. A noncache_code section is mapped to memory that cannot be configured as cache. The noncache_code section is used by the run-time library (RTL).

Data Storage

The data section, data1, is where the compiler puts global and static data in memory. The data section, constdata, is where the compiler puts data that has been declared as const. By default, the compiler places global zero-initialized data into a "BSS-style" section, called bsz, unless the compiler is invoked with the -no-bss switch (-no-bss). The cplb_data section exists so that configuration tables used to manage memory protection can be placed in memory areas that are always flagged as accessible.

Run-Time Stack

The run-time stack is positioned in memory section stack and is required for the run-time environment to function. The section must be mapped in the .ldf file.

The run-time stack is a 32-bit-wide structure, growing from high memory to low memory. The compiler uses the run-time stack as the storage area for local variables and return addresses. See Managing the Stack for more information.

Run-Time Heap Storage

The run-time heap section, heap, is where the compiler puts the run-time heap in memory. When linking, use your .ldf file to map the heap section. To dynamically allocate and deallocate memory at run-time, the standard C run-time library includes four functions: malloc(), calloc(), realloc(), and free().

Additionally, the C++ new and delete operators are available to allocate and free memory from the run-time heap. By default, all heap allocations are from the heap section of memory. The .ldf file must define symbolic constants ldf_heap_space, ldf_heap_end, and ldf_heap_length to allow the heap management routines to function.

Heap allocations may also be served from other memory regions. For more information, see Using Multiple Heaps.

Global Array Alignment

Global arrays must be aligned on a 32-bit word boundary or greater; the compiler will normally use this knowledge when optimizing accesses. If you declare arrays in assembly files that will be accessed from C/C++, use the .ALIGN directive to ensure the array's starting address has an alignment of 4 or greater.

Controlling System Heap Size and Placement

The system heap is the default heap used by calls to allocation functions like `malloc()` in C and the `new` operator in C++. System heap placement and size are specified in the application's `.ldf` file.

For details on adding and managing additional heaps besides the system heap, see Using Multiple Heaps.

Managing the System Heap in the IDE

The `.ldf` files created by the *Project Wizard,* with *Startup Code/LDF* option accepted, can be controlled using selections in the *System Configuration Overview* dialog box.

1. Expand your new project in a project navigation view such as *Project Explorer*.

2. Double-click `system.svc`. The *Startup Code/LDF* component appears in the *System Configuration Overview* dialog box.

3. Click the *Startup Code/LDF* tab at the bottom of the dialog box.

4. Click the *LDF* tab that appears at the left of the dialog box. The *LDF Configuration* page appears.

5. In the *System heap* area, check the Customize the system heap option.

6. You can now modify the size of the system heap, and choose into which memory it is placed.

7. When you have modified the settings as required, save the changes, via *Ctrl+S*, using File > Save, or by clicking on the floppy disk icon in the toolbar; this will cause the IDE to generate an updated LDF and related startup-code files, which will configure your heaps during the application's startup.

Managing the System Heap in the .ldf File

If an `.ldf` file has not been added to the project either by using the *Project Wizard* or by using a custom file, a default `.ldf` file from the `<install_path>\Blackfin\ldf` directory is used.

By default, the compiler uses the file `arch.ldf`, where `arch` is specified via the `-proc arch` switch. For example, if `-proc ADSP-BF537` is used, the compiler defaults to using `adsp-BF537.ldf`. The entry controlling the heap has a format similar to the following (which is simplified for clarity):

```
// macro that defines minimum system heap size
#define HEAP_SIZE 7K
L1_DATA
{
   INPUT_SECTION_ALIGN(4)
   // allocate minimum of HEAP_SIZE to system heap
   RESERVE(sys_heap, sys_heap_length = HEAP_SIZE, 4)
}> MEM_L1_DATA_A

// all other uses of MEM_L1_DATA_A

sys_heap
{
   INPUT_SECTION_ALIGN(4)
```

```
   // if any of MEM_L1_DATA_A is unused, add to system heap
   RESERVE_EXPAND(sys_heap, sys_heap_length, 0, 4)
   // define symbols to configure the heap for runtime support
   ldf_heap_space = sys_heap;
   ldf_heap_end = ldf_heap_space + sys_heap_length;
   ldf_heap_length = ldf_heap_end - ldf_heap_space;
}> MEM_L1_DATA_A
```

In this example, the minimal size of the heap can be modified by changing the definition of the `HEAP_SIZE` macro. If this value is larger than the memory output section being used, the linker issues error `li2040`.

The following macros can be used to configure the sizes of the system heap and stack, when using the default `.ldf` files. When using these macros, all three must be defined, for any of the definitions to take effect.

- `HEAP_SIZE` - Defines the size of the system heap. A typical value would be "7K".

- `STACK_SIZE` - Defines the size of the system stack. A typical value would be "8K".

- `STACKHEAP_SIZE` - Defines the size of the combined area used for system heap and system stack. A typical value would be "15K". Must be defined to be the sum of HEAP_SIZE and STACK_SIZE.

The default `.ldf` files support the placement of heaps in scratchpad (where available), L1, L2 (where available), or SDRAM. By default, L1 is used. To select alternate heap placement, the following macros can be defined when linking:

- `USE_SCRATCHPAD_HEAP` - Causes scratchpad memory to be used for the system heap. Limited to 4K capacity, but provides fast access and uses memory that might otherwise be unused. (This macro is not supported for Blackfin+ processors, where the scratchpad memory has been replaced with a third fully-featured L1 data memory block.)

- `USE_L1DATA_HEAP` - (default) Places the heap in L1 data block A

- `USE_L2_HEAP` - Causes L2 memory to be used for the system heap

- `USE_SDRAM_HEAP` - Causes SDRAM memory to be used for the system heap. It provides large capacity but is slow to access. Enabling data cache for the memory used reduces the performance impact.

Besides the default system heap, you can also define other heaps. See Using Multiple Heapsfor more information.

Standard Heap Interface

The standard functions, `calloc` and `malloc`, allocate a new object from the default heap. If `realloc` is called with a null pointer, it too allocates a new object from the default heap.

Previously allocated objects can be deallocated with the `free` or `realloc` functions. When a previously allocated object is resized with `realloc`, the returned object is in the same heap as the original object.

The `space_unused` function returns the number of bytes unallocated in the heap with index 0. Note that you may not be able to allocate all of this space due to heap fragmentation and the overhead that each allocated block needs.

Using Multiple Heaps

The Blackfin C/C++ run-time library supports the standard heap management functions `calloc`, `free`, `malloc`, and `realloc`. By default, a single heap, called the *default heap*, serves all allocation requests that do not explicitly specify an alternative heap. The default heap is defined in the standard linker description file and the run-time header.

Any number of additional heaps can be defined. These heaps serve allocation requests that are explicitly directed to them. These additional heaps can be accessed via the extension routines `heap_calloc`, `heap_free`, `heap_malloc`, and `heap_realloc`. For more information, see Using the Alternate Heap Interface.

Multiple heaps allow the programmer to serve allocations using fast-but-scarce memory or slower-but-plentiful memory as appropriate.

The following sections describe how to define a heap, work with heaps, use the heap interface, and free space in the heap.

Defining a Heap

Heaps can be defined in the IDE or at runtime. In both cases, a heap has three attributes:

- Start (base) address (the lowest usable address in the heap)

- Length (in bytes)

- User identifier (`userid`, a number >= 1)

The default system heap, defined at link-time, always has `userid 0`. In addition, heaps have indices. This is like the `userid`, except that the index is assigned by the system. All the allocation and deallocation routines use heap indices, not heap user IDs. A `userid` can be converted to its index using `heap_lookup()`. Be sure to pass the correct identifier to each function.

Defining Additional Heaps in the IDE

The Startup Code/LDF add-in allows you to configure and extend your heaps through a convenient graphical interface:

- Modify the size of your heaps.

- Change whether they are in internal or external memory (where available).

- Add additional heaps, or remove them.

To add a new heap:

1. Expand your new project in a project navigation view such as *Project Explorer*.

2. Double-click `system.svc`. The *Startup Code/LDF* component appears in the *System Configuration Overview* dialog box.

3. Click the *Startup Code/LDF* tab at the bottom of the dialog box.

4. Click the *LDF* tab that appears at the left of the dialog box. The *LDF Configuration* page appears.

5. In the *Stack and Heaps* area, click on *System heap*.

6. Click *Add....* The *Add User Heap* dialog box appears, and you can fill in the details of your new heap. Click *OK* when finished.

7. When you have modified the settings as required, save the changes, via *Ctrl+S*, using *File > Save*, or by clicking on the floppy disk icon in the toolbar; this will cause the IDE to generate an updated LDF and related startup-code files, which will configure your heaps during the application's startup.

The same interface allows you to edit additional heaps or remove them, via the *Edit...* and *Remove...* buttons, respectively.

Defining Heaps at Runtime

Heaps may also be defined and installed at runtime, using the `heap_install()` function:

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

This function can take any section of memory and start using it as a heap. It returns the heap index allocated for the newly installed heap, or a negative value if there was some problem. (See Tips for Working With Heaps.)

Reasons why `heap_install()` may return an error status include, but are not limited to:

- A heap using the specified `userid` already exists

- A new heap appears too small to be usable (length too small)

A heap is automatically initialized during installation. If necessary, a heap can be re-initialized later on. For more information, see Freeing Space.

Tips for Working With Heaps

Heaps may not start at address zero (`0x0000 0000`). This address is reserved and means "no memory could be allocated". It is the null pointer on the Blackfin processors.

Not all memory in a heap is available to users. 32 bytes per heap and 12 bytes per allocation (rounded to ensure the allocation is 8-byte aligned) are used for housekeeping. Thus, a heap of 256 bytes is unable to serve four blocks of 64 bytes.

Memory reserved for housekeeping precedes the allocated blocks. Thus, if a heap begins at `0x0800 0000`, this particular address is never returned to the user program as the result of an allocation request; the first request returns an address some way into the heap.

The base address of a heap must be appropriately aligned for an 8-byte memory access. This means that allocations can then be used for vector operations.

For C++ compliance, calls to `malloc` and `calloc` with a size of 0 allocate a block of size 1.

Allocating C++ STL Objects to a Non-Default Heap

C++ STL objects can be placed in a non-default heap through use of a custom allocator. To do this, you must first create your custom allocator. Below is an example custom allocator that you can use as a basis for your own. The most important part of customalloc.h in most cases is the allocate function, where memory is allocated to the STL object. Currently, the pertinent line of code assigns to the default heap (0):

```
Ty* ty = (Ty*) heap_malloc(0, n * sizeof(Ty));
```

Simply by changing the first parameter of heap_malloc(), you can allocate to a different heap:

- 0 is the default heap

- 1 is the first user heap

- 2 is the second user heap

- And so on

Once you have created your custom allocator, you must inform your STL object to use it. Note that the standard definition for "list":

```
list<int> a;
```

is the same as writing:

```
list<int, allocator<int> > a;
```

where "allocator" is the default allocator. Therefore, we can tell list "a" to use our custom allocator as follows:

```
list<int, customallocator<int> > a;
```

Once created, the list "a" can be used as normal. Also, example.cpp (below) is a simple example that shows the custom allocator being used.

customalloc.h

```
template <class Ty>
class customallocator {
public:
    typedef Ty value_type;
    typedef Ty* pointer;
    typedef Ty& reference;
    typedef const Ty* const_pointer;
    typedef const Ty& const_reference;

    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    template <class Other>
    struct rebind { typedef customallocator<Other> other; };
    pointer address(reference val) const { return &val; }
    const_pointer address(const_reference val)
    const { return &val; }
    customallocator(){}
```

```
   customallocator(const customallocator<Ty>&){}
   template class <Other>
   customallocator(const customallocator<Other>&) {}
   template <class Other>
   customallocator<Ty>& operator=(const customallocator&)
   { return (*this); }

   pointer allocate(size_type n, const void * = 0) {
   Ty* ty = (Ty*) heap_malloc(0, n * sizeof(Ty));
   cout << "Allocating 0x" << ty << endl;
   return ty;
}

void deallocate(void* p, size_type) {
   cout << "Deallocating 0x" << p << endl;
   if (p) free(p);
}

void construct(pointer p, const Ty& val) {
   new((void*)p)Ty(val); }
   void destroy(pointer p) { p->~Ty(); }
   size_type max_size() const { return size_t(-1); }
};
```

example.cpp

```
#include <iostream>
#include <list>
#include <customalloc.h>    // include your custom allocator
using namespace std;

main(){
   cout << "creating list" << endl;
   list <int, customallocator<int> > a;
                             // create list with custom allocator
   cout.setf(ios_base::hex,ios_base::basefield);
   cout << "pushing some items on the back" << endl;
   a.push_back(0xaaaaaaaa);   // push items as usual
   a.push_back(0xbbbbbbbb);
   while(!a.empty()){
      cout << "popping:0x" << a.front() << endl;
                             //read item as usual
      a.pop_front();         //pop items as usual
   }
   cout << "finished." << endl;
}
```

Using the Alternate Heap Interface

The C run-time library provides the alternate heap interface functions `heap_calloc`, `heap_free`, `heap_malloc`, and `heap_realloc`. These routines work in exactly the same way as the corresponding standard functions without the `heap_` prefix, except that they take an additional argument that specifies the heap index.

For example,

```
int heap_install(void base, size_t length, int userid);
int heap_init(int idx);
void *heap_calloc(int idx, size_t nelem, size_t elsize);
void *heap_free(int idx, void *);
void *heap_malloc(int idx, size_t length);
void *heap_realloc(int idx, void *, size_t length);
int heap_space_unused(int idx);
```

The actual entry point names for the alternate heap interface routines have an initial underscore. The `stdlib.h` standard header file defines macro equivalents without the leading underscores.

Note that for

```
heap_realloc(idx, NULL, length)
```

the operation is equivalent to

```
heap_malloc(idx, length)
```

However, for

```
heap_realloc(idx, ptr, length)
```

where `ptr != NULL`, the supplied `idx` parameter is ignored; the reallocation is always done from the heap that `ptr` is allocated from, even if a `memcpy` function is required within the heap.

Similarly,

```
heap_free(idx, ptr)
```

ignores the supplied index parameter, which is specified only for consistency-the space indicated by `ptr` is always returned to the heap from which it was allocated.

The `heap_space_unused(int idx)` function returns the number of bytes unallocated in the heap with index `idx`. The function returns −1 if there is no heap with the requested heap index.

C++ Run-Time Support for Alternate Heap Interface

The C++ run-time library provides support for allocation and release of memory from an alternative heap via the `new` and `delete` operators.

Heaps should be initialized with the C run-time functions as described. These heaps can then be used via the `new` and `delete` mechanism by simply passing the heap ID to the `new` operator. There is no need to pass the heap ID to the `delete` operator as the information is not required when the memory is released.

The routines are used as in the following example.

CCES 2.9.0 C/C++ Compiler and Library Manual for Blackfin Processors

```
#include <heapnew>

char *alloc_string(int size, int heapID)
{
    char *retVal = new(heapID) char[size];
    return retVal;
}

void free_string(char *aString)
{
    delete aString;
}
```

Freeing Space

When space is "freed", it is not returned to the "system". Instead, freed blocks are maintained on a free list within the heap in question. The blocks are coalesced where possible.

It is possible to re-initialize a heap, emptying the free list and returning all the space to the heap itself, using the `heap_init` function:

```
int heap_init(int index);
```

This returns zero for success, and nonzero for failure. Note, however, that this discards all records within the heap, so it may not be used if there are any live allocations on the heap still outstanding.

Startup and Termination

When the processor starts running, it somehow has to transfer control to the application's `main()` function, and it has to ensure that, before doing so, all the expected parts of the C/C++ run-time environment have been set up, including:

- Registers, which must be configured according to the rules in Registers.

- Heap and stack, which must be set up according to Controlling System Heap Size and Placementand Managing the Stack.

- Global variables must have been initialized to their starting values.

- Constructors of any static global instances must have been run.

- The arguments to `main()`, `argc` and `argv`, must have been set up.

This is the job of the startup code (also "C Run-Time Header" or "CRT"). The startup code is described in the *System Run-Time Documentation*, but some additional information is provided in the following sections.

- Memory Initialization

- Global Constructors

- Support for argv/argc

Memory Initialization

When control flow reaches the start of `main()`, global and static variables must have been initialized to their default values. When you build your application, the tool chain arranges for the executable image to contain sections of memory that are either zero- or value-filled, depending on how your data is declared. The image also contains sections that are filled with executable code. Further details are in Memory Section Usage.

During development, when you load your application into your processor using the IDE, the IDE copies the contents of those sections from your executable image into the processor's memory.

Once your application is complete, you have to change your application so that you no longer rely on using the IDE to load it into memory. In most cases, this can be done using the loader to create a bootable image that can be stored in non-volatile memory, such as a SPI flash, and loaded into memory at power-up by the Boot Code. In this model, the Boot Code arranges for all of your application's code and data sections to be copied into the final volatile memory space before control is transferred to your application. For details on this process, refer to the *Loader and Utilities Manual*, and to your processor's programming reference manual.

You can also make use of the memory initializer, a linker that can be enabled using the `-mem` switch (-mem). The memory initializer processes your executable image so that output sections marked as `RUNTIME_INIT` have their contents converted into an initialization stream stored in the `.meminit` section. This section, along with your application's startup code, is usually mapped to non-volatile memory.

In this model, when the Boot Code transfers control to your application, your application's code and data have not yet all been transferred to their final locations in volatile memory. Instead, the startup code (which is in non-volatile memory) invokes the `mi_initialize()` run-time library function, which processes the initialization stream. This performs the task of transferring your application's code and data to volatile memory.

For more details, refer to the *System Run-Time* Documentation and the *Linker and Utilities Manual*.

Global Constructors

The C/C++ run-time environment supports constructors and destructors.

Constructors and Destructors of Global Class Instances

Constructors for global class instances are invoked by the C/C++ run-time header during start-up. Several components allow this to happen

- The associated data space for the instance

- The associated constructor (and destructor, if one exists) for the class

- A compiler-generated "start" routine

- A compiler-generated table of such "start" routines

- A compiler-constructed linked-list of destructor routines

- The run-time header itself

The interaction of these components is as follows.

The compiler generates a "start" routine for each module that contains globally-scoped class instances that need constructing or destructing. There is at most one "start" routine per module; it handles all the globally-scoped class instances in the module:

- For each such instance, it invokes the instance's constructor. This may be a direct call, or it may be inlined by the compiler optimizer.

- If the instance requires destruction, the "start" routine registers this fact for later, by including pointers to the instance and its destructor into a linked list.

The start routine is named after the first such instance encountered, though the classes are not guaranteed to be constructed or destructed in any particular order (with the exception that destructors are called in the reverse order of the constructors). Such instances should not have any dependency on construction order; the `-check-init-order` switch (-check-init-order) is useful for verifying this during system development, as it plants additional code to detect uses of unconstructed objects during initialization.

A pointer to the "start" routine is placed into the `ctor` section of the generated object file. When the application is linked, all `ctor` sections are mapped into the same `ctor` output section, forming a table of pointers to the "start" routines. An additional `ctorl` object is appended to the end of the table; this contains a terminating NULL pointer.

When the run-time header is invoked, it calls `_ctor_loop()`, which walks the table of `ctor` sections, calling each pointed-to "start" function until it reaches the NULL pointer from `ctorl`. In this manner, the run-time header calls each global class instance's constructor, indirectly through the pointers to "start" functions.

When the program reaches `exit()`, either by calling it directly or by returning from `main()`, the `exit()` routine follows the normal process of invoking the list of functions registered through the `atexit()` interface. One of these is a function that walks the list of destructors, invoking each in turn (in reverse order from the constructors).

This function is registered with `atexit()` via `_mark_dtors()`; the compiler plants a call to this function at the start of every `main()` that is compiled in C++ mode.

> **NOTE:** Functions registered with `atexit()` may not reference global class instances, as the destructor for the instance may be invoked before the reference is used.

Constructors, Destructors, and Memory Placement

By default, the compiler places the code for constructors and destructors into the same section as any other function's code. This can be changed either by specifying the section specifically for the constructor or destructor (see #pragma section/#pragma default_section and Placement Support Keyword (section)), or by altering the default destination section for generated code (see #pragma section/#pragma default_section and -section `id=section_name[, id=section_name...]`. If a constructor is inlined into the "start" routine by the optimizer, such placement will have no effect. For more information, see Inlining and Sections.

While normal compiler-generated code is placed into the `CODE` area, the "start" routine is placed into the `STI` area. Both `CODE` and `STI` default to the same section, but may be changed separately using `#pragma default_section` or the `-section` switch (since the "start" function is an internal function generated by the compiler, its placement cannot be affected by `#pragma section`).

The pointer to the "start" routine is placed into the `ctor` section. This is not configurable, as the invocation process relies on all of the "start" routine pointers being in the same section during linking, so that they form a table. It is essential that all relevant `ctor` sections are mapped during linking; if a `ctor` section is omitted, the associated constructor will not be invoked during start-up, and run-time behavior will be incorrect.

If destructors are required, the compiler generates data structures pointing to the class instance and destructor. These structures are placed into the default variable-data section (the `DATA` area).

Support for argv/argc

By default, the facility to specify arguments that are passed to your `main()` (`argv/argc`) at run-time is enabled. However, to correctly set up `argc` and `argv` requires additional configuration by the user. Modify your application as follows:

Define your command-line arguments in C by defining a variable called `__argv_string`. When linked, your new definition overrides the default zero definition otherwise found in the C run-time library.

For example,

```
extern const char __argv_string[] = "prog_name -in x.gif -out y.jpeg";
```

Compiler C++ Template Support

The compiler provides C++ template support as defined in the ISO/IEC 14882:2003 standard, or the ISO/IEC 14882:2011 standard (if enabled with the -c++11 switch).

Template Instantiation

Templates are instantiated automatically by the prelinker during compilation (see Compiler Components). This involves compiling files, determining any required template instantiations, and then recompiling those files making the appropriate instantiations. The process repeats until all required instantiations have been made. Multiple recompilations may be required in the case when a template instantiation is made that requires another template instantiation to be made.

Exported templates and implicit instantiation are two approches that allow separating template declarations from their implementations. However, it should be noted that unlike with functions, code that uses such a template will still depend on the template implementation, which means that the code needs to be recompiled if the template implementation is changed.

Exported Templates

The compiler supports exported templates as per the ISO/IEC 14482:2003 standard. Exported templates have been removed from the ISO/IEC 14492:2011 standard, and hence are not supported when -c++11 mode is enabled.

An exported template does not need to be present in a translation unit that uses the template. For example, the following is a valid C++ program consisting of two translation units:

```
// File 1
#include <iostream>
```

```
static void print(void) { std::cout << "File 1" << std::endl;}
export template <class T> T const &maxii(T const &a, T const &b);
int main()
{
    print();
    return maxii(7,8);
}

// File 2
#include <iostream>

static void print(void) { std::cout << "File 2" << std::endl;}
export template <class T> T const &maxii(T const &a, T const &b)
{
    print();
    return (a>b) ? a : b;
}
```

The first file makes use of the `maxii()` function exported by the second. Unrelated to this, both files declare their own, private copy of the `print()` function.

The two files are separate translation units; one is not included in the other, so no linking errors arise due to the individual definitions of the `print()` functions.

If `file1.c` obtained `file2.c`'s definition of `maxii()` by including `file2.c` into `file1.c` (whether explicitly or implicitly - see Implicit Instantiation), `file1.c` would also include `file2.c`'s definition of the `print()` function, leading to a linkage error.

When a file containing a definition of an exported template is compiled, a file with a `.et` suffix is created and some extra information is included in the associated `.ti` file. The `.et` files are used by the compiler to find the translation units that define a given exported template.

Implicit Instantiation

As an alternative to Exported Templates, the compiler can use a method called *implicit instantiation*, which is common practice. It results in having both the specification and definition available at the point of instantiation.

> **NOTE:** Implicit instantiation does not conform to the C++ standards and does not work with exported templates. Implicit instantiation is disabled by default. It can be enabled via the -implicit-inclusion switch.

Implicit instantiation involves placing template specifications in a header (for example, `.h`) file and the definitions in a source (for example, `.cpp`) file. Any file being compiled that includes a header file containing template specifications instructs the compiler to implicitly include the corresponding `.cpp` file containing the definitions of the compiler.

For example, you may have the header file `tp.h`:

```
template <typename A> void func(A var);
```

and source file `tp.cpp`:

```
template <typename A> void func(A var)
{
    ...code...
}
```

Two files `file1.cpp` and `file2.cpp` that include `tp.h` have file `tp.cpp` included implicitly to make the template definitions available to the compilation.

When generating dependencies, the compiler will only parse each implicitly included `.cpp` file once. This parsing avoids excessive compilation times in situations where a header file that implicitly includes a source file is included several times. If the `.cpp` file should be included implicitly more than once, the `-full-dependency-inclusion` switch ([-full-dependency-inclusion](#)) can be used. (For example, the file may contain macro guarded sections of code.) This may result in more time required to generate dependencies.

Generated Template Files

Regardless of whether implicit instantiation is used, the compilation process involves compiling one or more source files and generating a `.ti` file corresponding to the source files being compiled. These `.ti` files are then used by the prelinker to determine the templates to be instantiated. The prelinker creates a `.ii` file and recompiles one or more of the files instantiating the required templates.

The prelinker ensures that only one instantiation of a particular template is generated across all objects. For example, the prelinker ensures that if both `file1.cpp` and `file2.cpp` invoked the template function with an `int`, the resulting instantiation would be generated in just one of the objects.

Identifying Un-Instantiated Templates

If for some reason the prelinker is unable to instantiate all the templates that are required for a particular link, then a link error occurs. For example,

```
[Error li1021] The following symbols referenced in processor 'P0'
        could not be resolved:
        'Complex<T1> Complex<T1>::_conjugate() const [with T1=short]
        [_conjugate__16Complex__tm__2_sCFv_18Complex__tm__4_Z1Z]' referenced
        from '.\Debug\main.doj'
        'T1 *Buffer<T1>::_getAddress() const [with T1=Complex<short>]
        [_getAddress__33Buffer__tm__19_16Complex__tm__2_sCFv_PZ1Z]'  referenced
        from '.\Debug\main.doj'
        'T1 Complex<T1>::_getReal() const [with T1=short]
        [_getReal__16Complex__tm__2_sCFv_Z1Z]' referenced from  '.\Debug
\main.doj'
Linker finished with 1 error
```

Careful examination of the linker errors reveals which instantiations have not been made. Below are some examples.

```
Missing instantiation:
        Complex<short> Complex<short>::conjugate()
Linker Text:
        'Complex<T1> Complex<T1>::_conjugate() const [with T1=short]
        [_conjugate__16Complex__tm__2_sCFv_18Complex__tm__4_Z1Z]'
        referenced from '.\Debug\main.doj'
```

```
Missing instantiation:
        Complex<short> *Buffer<Complex<short>>::getAddress()
Linker Text:
        'T1 *Buffer<T1>::_getAddress() const [with T1=Complex<short>]
        [_getAddress__33Buffer__tm__19_16Complex__tm__2_sCFv_PZ1Z]'
         referenced from '.\Debug\main.doj'
Missing instantiation:
        Short Complex<short>::getReal()
Linker Text:
        'T1 Complex<T1>::_getReal() const [with T1=short]
        [_getReal__16Complex__tm__2_sCFv_Z1Z]' referenced from
        '.\Debug\main.doj'
```

There could be many reasons for the prelinker being unable to instantiate these templates, but the most common is that the .ti and .ii files associated with an object file have been removed. Only source files that can contain instantiated templates have associated .ti and .ii files, and without this information, the prelinker may not be able to complete its task. Removing the object file and recompiling will normally fix this problem.

Another possible reason for un-instantiated templates at link time is when implicit inclusion (described above) is disabled but the source code has been written to require it. If -implicit-inclusion is not to be enabled, this can be fixed by explicitly including the .cpp files that would have been implicitly included into their corresponding headers.

Another likely reason for seeing the linker errors above is invoking the linker directly. It is the compiler's responsibility to instantiate C++ templates, and this is done automatically if the final link is performed via the compiler driver. The linker itself contains no support for instantiating templates.

File Attributes

A *file attribute* is a name-value pair that is associated with a binary object, whether in an object file (.doj) or in a library file (.dlb). One attribute name can have multiple values associated with it. Attribute names and values are strings. A valid attribute name consists of one or more characters matching the following pattern:

[a-zA-Z_][a-zA-Z_0-9]*

An attribute value is a non-empty character sequence containing any characters apart from NUL.

Attributes help with the placement of run-time library functions. All of the run-time library objects contain attributes that allow you to place time-critical library objects into internal (fast) memory. Using attribute filters in the .ldf file, you can place run-time library objects into internal or external (slow) memory, either individually or in groups.

The run-time libraries have attributes associated with the objects in them. For more information, see Library Attributes in the *DSP Run-Time Library* chapter.

This section also describes:

• Automatically-Applied Attributes

- Default LDF Placement

- Sections Versus Attributes

- Using Attributes

Automatically-Applied Attributes

By default, the compiler applies a number of attributes automatically when compiling a C/C++ file. For example, it applies the `Content` and `FuncName` attributes . These automatically-applied attributes can be disabled using the (-no-auto-attrs) switch.

The `Content` attribute can be used to map binary objects according to their kind of content, as show by the *Interpreting Values of the Content Attribute* table. The *Content Attributes* figure shows a content attribute tree.
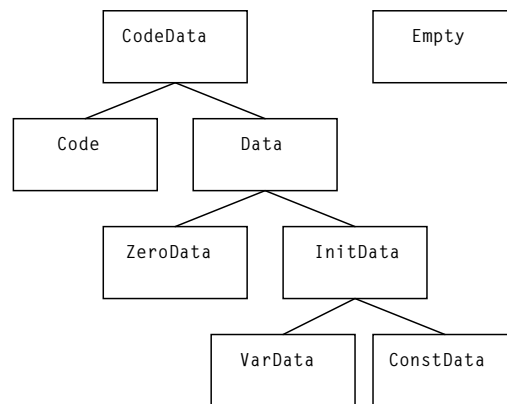
Figure 2-10: Content Attributes

Table 2-50: Interpreting Values of the Content Attribute

`CodeData`	This is the most general value, indicating that the binary object contains a mix of content types.
`Code`	The binary object does not contain any global data, only executable code. This can be used to map binary objects into program memory, or into ROM.
`Data`	The binary object does not contain any executable code. The binary object may not be mapped into dedicated program memory. The kinds of data used in the binary object vary.
`ZeroData`	The binary object contains only zero-initialized data. Its contents must be mapped into a memory section with the `ZERO_INIT` qualifier, to ensure correct initialization.
`InitData`	The binary object contains only initialized global data. The contents may not be mapped into a memory section that has the `ZERO_INIT` qualifier.
`VarData`	The binary object contains initialized variable data. It must be mapped into read-write memory, and may not be mapped into a memory section with the `ZERO_INIT` qualifier.
`ConstData`	The binary object contains only constant data (data declared with the C const qualifier). The data may be mapped into read-only memory (but see also the `-const-read-write` switch (-const-read-write) and its effects.

Table 2-50: Interpreting Values of the Content Attribute (Continued)

`Empty`	The binary object contains neither functions nor global data.

Default LDF Placement

The default `.ldf` file is written in such manner that the order of preference for putting an object in section `data1` or `program` depends on the value of the `prefersMem` attribute. Precedence is given in the following order:

1. Highest priority is given to binary objects that have a `prefersMem` attribute with a value of `internal`.

2. Next priority is given to binary objects that have no `prefersMem` attribute, or a `prefersMem` attribute with a value that is neither `internal` nor `external`.

3. Lowest priority is given to binary objects with a `prefersMem` attribute with the value `external`.

Although the default `.ldf` files only reference the values `internal` and `external`, `prefersMem` may have other values. For example, an object using a value, such as `L2`, is given second priority, as the value is neither `internal` nor `external`. You may modify your `.ldf` file to assign appropriate priority to any value you choose, by mapping objects with higher-priority values before objects with lower-priority values.

The `prefersMemNum` attribute is similar to the `prefersMem` attribute, but is given numerical values instead of textual values. This makes it easier to assign priority when there are many different levels, because you can use relational comparisons in the `.ldf` file instead of just equalities and inequalities. The ***Values for prefersMemNum Attribute*** table shows the numerical values used by the run-time library for each corresponding `prefersMem` attribute value.

Table 2-51: Values for prefersMemNum Attribute

prefersMem Attribute Value	*prefersMemNum Attribute Value*
internal	30
any	50
external	70

Sections Versus Attributes

File attributes and section qualifiers ([Placement Support Keyword (section)](#)) can be thought of as being somewhat similar, since both affect how the application is linked. There are important differences, however, that affect whether you choose to use sections or file attributes to control the placement of code and data.

Granularity

Individual components-global variables and functions-in a binary object can be assigned different sections, and then those section assignments can be used to map each component of the binary object differently. In contrast, an attribute applies to the whole binary object. This means you do not have as fine control over individual components using attributes as when using sections.

Hard Mapping Versus Soft Mapping

A section qualifier is a " hard" constraint. When the linker maps the object file into memory, it must obey all the section qualifiers in the object file, according to instructions in the .ldf file. If this cannot be done, or if the .ldf file does not give sufficient information to map a section from the object file, the linker reports an error.

In contrast, with attributes, the mapping is "soft". The default .ldf files use the prefersMem attribute as a guide to give a better mapping in memory, but if this cannot be done, the linker will not report an error. For example, if there are more objects with prefersMem=internal than will fit into internal memory, the remaining objects will spill over into external memory. Likewise, if there are fewer objects with the attribute prefersMem!=external than are needed to fill internal memory, some objects with the attribute prefersMem=external may be mapped to internal memory.

Section qualifiers are rules that must be obeyed. Attributes are guidelines, defined by convention, that can be used if convenient and ignored if not. The Content attribute is an example of this: you can use the Content attribute to map Code and ConstData binary objects into read-only memory, if this is a convenient partitioning of your application, but you need not do so if you choose to map your application differently.

Number of Values

Any given element of an object file is assigned exactly one section qualifier, to determine into which section it should be mapped. In contrast, an object file may have many attributes (or even none), and each attribute may have many different values. Since attributes are optional and act as guidelines, you need only pay attention to the attributes that are relevant to your application.

Using Attributes

You can add attributes to a file in two ways:

- Use #pragma file_attr (#pragma file_attr("name[=value]" [, "name[=value]" [...]]))

- Use the -file-attr switch (-file-attr *name[=value]*)

The run-time libraries have attributes associated with the objects in them. For more information, see Library Attributes in the *DSP Run-Time Library* chapter.

Example 1

This example uses attributes to encourage the placement of library functions in internal memory.

Suppose the file test.c exists, as shown below:

```
#define MANY_ITERATIONS 500

void main(void) {
   int i;
   for (i = 0; i  MANY_ITERATIONS; i++) {
      fft_lib_function();
      frequently_called_lib_function();
   }
```

```
    rarely_called_lib_function();
}
```

Also suppose:

- The objects containing `frequently_called_lib_function` and `rarely_called_lib_function` are both in the standard library, and have the attribute `prefersMem=any`.

- There is only enough internal memory to map `fft_lib_function` (which has `prefersMem=internal`) and one other library function into internal memory.

- The linker chooses to map `rarely_called_lib_function` to internal memory.

In this example, for optimal performance, `frequently_called_lib_function` should be mapped to the internal memory in preference to `rarely_called_lib_function`.

The `.ldf` file defines a macro `$OBJS_LIBS_INTERNAL` to store all the objects that the linker should try to map to internal memory, as follows:

```
$OBJS_LIBS_INTERNAL = $OBJECTS{prefersMem("internal")},
$LIBRARIES{prefersMem("internal")};
```

If all the objects do not fit in internal memory, the remainder is placed in external memory and no linker error will occur. To add the object that contains `frequently_called_lib_function` to this macro, extend the definition to read:

```
$OBJS_LIBS_INTERNAL =
    $OBJECTS{prefersMem("internal")},
    $LIBRARIES{prefersMem("internal")},
    $LIBRARIES{ libFunc("frequently_called_lib_function") };
```

This ensures that the binary object that defines `frequently_called_lib_function` is among those to which the linker gives highest priority when mapping binary objects to internal memory.

Note that it is not necessary to know which binary object (or even which library) defines `frequently_called_lib_function`. All the binary objects in the run-time libraries define the `libFunc` attribute so that you can select the binary objects for particular functions without needing to know exactly where in the libraries a function is defined. The modified line uses this attribute to select the binary object (or objects) for `frequently_called_lib_function` and append it (or them) to the `$OBJS_LIBS_INTERNAL` macro. The `.ldf` file maps objects in `$OBJS_LIBS_INTERNAL` to internal memory in preference to other objects, so `frequently_called_lib_function` is mapped to L1.

For more information, see Library Attributes in the *C/C++ Run-Time Library* chapter.

Example 2

Suppose you want the contents of `test.c` to be mapped to external memory by preference. You can do this by adding the following pragma to the top of `test.c`:

```
#pragma file_attr("prefersMem=external")
```

or use the `-file-attr` switch:

```
ccblkfn -file-attr prefersMem=external test.c
```

Both methods cause the resulting object file to have the attribute `prefersMem=external`. The `.ldf` files give objects with this attribute the lowest priority when mapping objects into internal memory, so the object is less likely to consume valuable internal memory space, which could be more usefully allocated to another function.

> **NOTE:** Since file attributes are used as guidelines rather than rules, if space is available in internal memory after higher-priority objects have been mapped, it is permissible for objects with `prefersMem=external` to be mapped into internal memory.

Implementation Defined Behavior

Each of the language standards supported by the compiler have implementation defined behavior for a list of areas. The implementation used by the CCES compilers is detailed in this section.

Enumeration Type Implementation Details

The CCES compiler by default implements the underlying type for enumerations as the first type from the following list that can be used to represent all the values in the specified enumeration : `unsigned int`, `int`, `unsigned long`, `long`, `unsigned long long`, `long long`. Enumeration constant values can be any integral type including `long long` and `unsigned long long`.

Enumerations types being implemented as `long long` or `unsigned long long` types is an Analog Devices extension to ANSI C89 standard (ISO/IEC 9899:1990). Allowing enumerations constants to be integral types other than int is an Analog Devices extension to the ANSI C89 and ANSI C99 (ISO/IEC 9899:1999) standards. These extensions can be disabled by using the `-enum-is-int` switch. For more information, see -enum-is-int.

When `-enum-is-int` is used the compiler issues error cc0066 "enumeration value is out of "int" range" when it encounters enumeration constant values that cannot be held using an int type. Warning cc1661 "enumeration value is greater than int type" is issued when larger than int type enumeration values are used and not compiling with the `-enum-is-int` switch.

The different underlying types used by the compiler to implement enumerations can give rise to other compiler warnings. For example in the following enumeration the underlying type will be unsigned int which will result in warning cc0186 "pointless comparison of unsigned integer with zero".

If a negative enumeration constant was added to the definition of *e1* or if the example was compiled with the `-enum-is-int` switch the underlying type used will be signed int and there would be no warning issued for the comparison.

```
typedef enum { v1, v2 } e1;
void check (e1 v) {
  if (v  0) /* pointless comparison if e1 is unsigned */
```

```
    printf("out of range");
}
```

ISO/IEC 9899:1990 C Standard (C89 Mode)

The contents of this section refer to Annex G of the ISO/IEC 9899:1990 C Standard; subsection numbers such as 5.1.1.3 refer to the relevant section of that Standard, which has some implementation-defined aspect.

G3.1 Translation

5.1.1.3 How a diagnostic is identified

The compiler will emit descriptive diagnostics via the standard error stream at compile time (e.g. "cc0223: function declared implicitly") or as annotations in generated assembly files.

G3.2 Environment

5.1.2.2.1 The semantics of the arguments to main

By default, `argv[0]` is a `NULL` pointer.

The values given to the strings pointed to be the `argv` argument can be defined by the user. For more information, see Support for argv/argc.

5.1.2.3 What constitutes an interactive device

An interactive device is considered a paired display screen and keyboard.

G3.3 Identifiers

6.1.2 The number of significant initial characters (beyond 31) in an identifier without external linkage

The number of significant initial characters in an identifier without external linkage is 15,000.

6.1.2 The number of significant initial characters (beyond 6) in an identifier with external linkage

Identifiers with external linkage are treated in the same way as identifiers without.

6.1.2 Whether case distinctions are significant in an identifier with external linkage

Case distinctions are significant.

G3.4 Characters

5.2.1 The members of the source and execution character sets, except as explicitly specified in this International Standard

The compiler supports the non-standard characters "$" and "`" (ASCII 39).

5.2.1.2 The shift states used for the encoding of multi-byte characters

No shift states are used for the encoding of multi-byte characters.

5.2.4.2.1 The number of bits in a character in the execution character set

8 Bits.

6.1.3.4 The mapping of members of the source character set (in character constants and string literals) to members of the execution character set

Characters in the source file are interpreted as ASCII values, which are also used in the execution environment.

6.1.3.4 The value of an integer character constant that contains a character or escape sequence not represented in the basic execution set or the extended character set for a wide character constant

An unrecognized escape sequence will have the escape character dropped. E.g. '\k' becomes 'k'.

6.1.3.4 The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multi-byte character

An integer character constant may contain up to 4 characters. If the constant contains between 2 and 4 characters, warning cc1994 will be issued. Using more than 4 characters will result in warning cc2226 being issued and all but the last 4 characters being discarded.

Where a wide character contains more than one multi-byte character, only the first character is retained and warning cc0026 will be issued. Subsequent characters are discarded.

6.1.3.4 The current locale used to convert multi-byte characters into corresponding wide characters (codes) for a wide character constant

Only the "C" locale is supported in Analog Devices tool chain and processors.

6.2.1.1 Whether a "plain" char has the same range of values as signed char or unsigned char

A "plain" `char` has the same range and value as a `signed char`; except on Blackfin, when the `-unsigned-char` switch is used (-unsigned-char).

G3.5 Integers

6.1.2.5 The representations and sets of values of the various types of integers

The representation is shown in the *Representations of Integer Types* table.

Table 2-52: Representations of Integer Types

Type	Width	Minimum Value	Maximum Value
(signed) char	8 bits	-128	127
unsigned char	8 bits	0	255
(signed) short	16 bits	-32768	32767
unsigned short	16 bits	0	65535
(signed) int	32 bits	-2147483648	2147483647
unsigned int	32 bits	0	4294967295

Table 2-52: Representations of Integer Types (Continued)

Type	Width	Minimum Value	Maximum Value
(signed) long	32 bits	-2147483648	2147483647
unsigned long	32 bits	0	4294967295
(signed) long long	64 bits	-9223372036854775808	9223372036854775807
unsigned long long	64 bits	0	18446744073709551615

6.2.1.2 The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented

When converting an unsigned integer to a signed integer of equal length, the exact value of the unsigned integer will be copied to the signed integer. If the sign bit is set, this will result in a negative number.

When converting a signed integer to a smaller signed integer, the lower bits of the signed integer (of the size of the smaller signed integer) are copied to the smaller signed integer. If the top-most copied bit is set, this will result in a negative number.

6.3 The results of bitwise operations on signed integers

The results of the operations are shown in the *Bitwise Operations on Signed Integers* table.

Table 2-53: Bitwise Operations on Signed Integers

~	Same as unsigned integer
<<	Same as unsigned integer
>>	Will fill upper bits with ones if sign bit was originally set
&	Same as unsigned integer
^	Same as unsigned integer
|	Same as unsigned integer

6.3.5 The sign of the remainder on integer division

The sign of the remainder on integer division will be the same as the sign of the first operand of the remainder operation.

6.3.7 The result of a right shift of a negative-valued signed integral type

Right shifts will retain the sign bit on a signed integer. All other bitwise operations treat signed integers as unsigned.

G3.6 Floating-Point

6.1.2.5 The representations and sets of values of the various types of floating-point numbers

The representations and value ranges are:

- `float`

- 32 bits (1 sign bit, 8 exponent bits, 32 mantissa bits)

    -3.4028234663852886E+38 to 3.4028234663852886E+38

- `double` (default setting)

    - 32 bits (1 sign bit, 8 exponent bits, 32 mantissa bits)

        -3.4028234663852886E+38 to 3.4028234663852886E+38

- `double` (when compiling with "`-double-size-64`")

    - 64 bits (1 sign bit, 11 exponent bits, 52 mantissa bits)

        -1.797693134862315708e+308 to 1.797693134862315708e+308

- `long double`

    - 64 bits (1 sign bit, 11 exponent bits, 52 mantissa bits)

        -1.797693134862315708e+308 to 1.797693134862315708e+308

6.2.1.3 The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value

Round to nearest, ties to even.

6.2.1.4 The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number

Round to nearest, ties to even.

G3.7 Arrays and Pointers

6.3.3.4, 7.1.1 The type of integer required to hold the maximum size of an array-that is, the type of the sizeof operator, size_t

```
long unsigned int;
```

6.3.4 The result of casting a pointer to an integer or vice-versa

A cast from pointer to integer results in the most-significant bits being discarded if the size of the pointer is larger than the integer. If the pointer is smaller than the integer type being cast to, the integer will be zero extended.

A cast from integer to pointer results in the most-significant bits being discarded if the size of the integer is larger that the pointer. If the integer is smaller than the pointer type being cast to, the pointer will be sign-extended.

6.3.6, 7.1.1 The type of integer required to hold the difference between two pointers to elements of the same array, ptrdiff_t

```
long int;
```

G3.8 Registers

6.5.1 The extent to which objects can actually be placed in registers by use of the register storage-class specifier

The `register` storage class specifier is ignored.

G3.9 Structures, Unions, Enumerations and Bit-Fields

6.3.2.3 A member of a union object is accessed using a member of a different type

The data stored in the appropriate location is interpreted as the type of the member accessed.

6.5.2.1 The padding and alignment of members of structures. This should present no problem unless binary data written by one implementation are read by another.

Within a structure, members of the fundamental types are aligned on a multiple of their size. Structures are aligned on the strictest alignment of any of their members, but are always aligned to at least 32 bits.

6.5.2.1 Whether a "plain" int bit-field is treated as a signed int bit-field or as an unsigned int bit-field

A "plain" `int` bit-field is treated as a `signed int` bit-field (including bit-fields of size 1).

6.5.2.1 The order of allocation of bit-fields within a unit

Low to High Order.

6.5.2.1 Whether a bit-field can straddle a storage-unit boundary

A bit-field will be placed in an adjacent storage unit instead of overlapping.

6.5.2.2 The integer type chosen to represent the values of an enumeration type

By default, the compiler defines enumeration types with integral types larger than `int`, if `int` is insufficient to represent all the values in the enumeration. The compiler can be forced to use only `int` through the use of the `-enum-is-int` switch (-enum-is-int).

G3.10 Qualifiers

6.5.3 What constitutes an access to an object that has volatile-qualified type

Any reference to a `volatile`-qualified object is considered to constitute an access.

G3.11 Declarators

6.5.4 The maximum number of declarators that may modify an arithmetic, structure, or union type

No maximum limit is enforced.

G3.12 Statements

6.6.4.2 The maximum number of case values in a switch statement

There is no hard-coded maximum number of `case` values in a `switch` statement.

G3.13 Preprocessing Directives

6.8.1 Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set; whether such a character constant may have a negative value

The character set used is the same.

Negative values are allowed.

6.8.2 The method for locating includable source files

Include files, whose names are not absolute path names and that are enclosed in "..." when included, are searched for in the following directories in this order:

1. The directory containing the current input file (the primary source file or the file containing the `#include`)

2. Any directories specified with the `-I` switch (`-I directory[{,|;} directory...]`) in the order they are listed on the command line

3. Any directories on the standard list: `<install_path>\...\include`

NOTE: If a file is included using the `<...>` form, this file is only searched for by using directories defined in items 2 and 3 above.

6.8.2 The support of quoted names for includable source files

Quoted file names are supported.

6.8.2 The mapping of source file character sequences

The source file character sequence is mapped to its corresponding ASCII character sequence.

6.8.6 The behavior on each recognized #pragma directive

For more information, see Pragmas.

6.8.8 The definitions for __DATE__ and __TIME__ when respectively, the data and time of translation are not available

The macros `__DATE__` and `__TIME__` will be defined as "[date unknown]" and "[time unknown]" respectively.

G3.14 Library Functions

7.1.6 The null pointer constant to which the macro NULL expands

`NULL` expands to `0`.

7.2 The diagnostic printed by and the termination behavior of the assert function

{file name}:{line number} {failed assertion expression} - Runtime Assertion.

7.3.1 The sets of characters tested for by the isalnum, isalpha, iscntrl, islower, isprint, and isupper functions

The following characters are tested:

- `isalnum` - 0-9, a-z or A-Z

- `isalpha` - a-z or A-Z

- `iscntrl` - 0x00-0x1F or 0x7F

- `islower` - a-z

- `isprint` - 0x20-0x7E

- `isupper` - A-Z

7.5.1 The values returned by the mathematics functions on domain errors

The values are:

- `acos`: 0

- `asin`: 0

- `atan2`: 0

- `log`: `-HUGE_VAL`

- `log10`: `-HUGE_VAL`

- `pow`: when the first parameter is 0 and the second is not an integral value, it returns 0. when the first parameter is zero and the second is less than zero, it returns `HUGE_VAL`.

- `sqrt`: 0

- `fmod`: 0

7.5.1 Whether the mathematics functions set the integer expression errno to the value of the macro ERANGE on underflow range errors

The state of `errno` should not be relied upon unless stated explicitly in the documentation.

7.5.6.4 Whether a domain error occurs or zero is returned when the fmod function has a second argument of zero

Zero is returned.

7.7.1.1 The set of signals for the signal function

The following signals are supported:

- `SIGTERM`

- `SIGABRT`

- `SIGFPE`

- `SIGILL`

- `SIGINT`

- `SIGSEGV`

7.7.1.1 The semantics for each signal recognized by the signal function

After the handler is invoked, the disposition of the signal is not reset to `SIG_DFL`.

7.7.1.1 The default handling and the handling at program startup for each signal recognized by the signal function

By default, `SIGABRT` will cause the program to terminate. All other signals are ignored by default.

7.7.1.1 If the equivalent of signal(sig, SIG_DFL); is not executed prior to the call of a signal handler, the blocking of the signal that is performed

Blocking of signals is not performed prior to the call of the signal handler.

7.7.1.1 Whether the default handling is reset if the SIGILL signal is received by a handler specified to the signal function

If the `SIGILL` signal is received, the reset to `SIG_DFL` is not performed.

7.9.2 Whether the last line of a text stream requires a terminating new-line character

The last line should have a terminating new-line character.

7.9.2 Whether space characters that are written out to a text stream immediately before a new-line character appear when read in

The space characters will appear.

7.9.2 The number of null characters that may be appended to data written to a binary stream

Any number of null characters may be appended.

7.9.3 Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file

End of the file.

7.9.3 Whether a write on a text stream causes the associated file to be truncated beyond that point

The file will become truncated.

7.9.3 The characteristics of file buffering

`stderr` is unbuffered, `stdio` is line-buffered, and other streams are fully buffered.

7.9.3 Whether a zero-length file actually exists

A zero-length file does exist.

7.9.3 The rule for composing valid file names

Any basic ASCII character that isn't reserved by the file system is valid.

7.9.3 Whether the same file can be open multiple times

A file can be opened multiple times.

7.9.4.1 The effect of the remove function on an open file

There will be no effect on the file and the function returns -1.

7.9.4.2 The effect if a file with the new name exists prior to a call to the rename function

There will be no effect on the files and the function returns -1.

7.9.6.1 The output for %p conversion in the fprintf function

The pointer address will be printed as an 8-character hexadecimal value. e.g. 00004010.

7.9.6.2 The input for %p conversion in the fscanf function

All valid values that can be interpreted as a hexadecimal value will be read until an invalid value or line break is reached, at which point no further characters are read. if the value is larger than can be stored in an 8-character hexadecimal, then the value will saturate.

7.9.6.2 The interpretation of a - character that is neither the first nor the last character in the scanlist for %[conversion in the fscanf function

A hyphen does not infer an inclusive range of values. e.g. `%[0-9]` will look for a sequence of '0', '-' and '5' chars.

7.9.9.1, 7.9.9.4 The value to which the macro errno is set by the fgetpos or ftell function on failure

`errno` should never be relied upon.

7.9.10.4 The messages generated by the perror function

`errno` should never be relied upon, so the error messages returned by this function should not be relied upon.

7.10.3 The behavior of the calloc, malloc, or realloc function if the size requested is zero

This is equivalent to a size request of 1.

7.10.4.1 The behavior of the abort function with regard to open and temporary files

`abort` will cause execution to jump to `exit` as if the program had run to the end of `main`.

7.10.4.3 The status returned by the exit function if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE

The `exit` function never returns.

7.10.4.4 The set of environment names and the method for altering the environment list used by the getenv function

The `getenv` function always returns `NULL`.

7.10.4.5 The contents and mode of execution of the string by the system function

The `system` function always returns 0 and has no effect.

7.11.6.2 The contents of the error message strings returned by the strerror function

"error #" followed by the number passed in.

7.12.1 The local time zone and Daylight Saving Time

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).

7.12.2.1 The era for the clock function

The era for the clock is the number of clock ticks since the start of program execution.

ISO/IEC 9899:1999 C Standard (C99 Mode)

The contents of this section refer to Annex J of the ISO/IEC 9899:1999 C Standard; the subsection numbers refer to parts of that Standard which have implementation-defined aspects.

J3.1 Translation

3.10, 5.1.1.3 How a diagnostic is identified

The compiler will emit descriptive diagnostics via the standard error stream at compile time (e.g. "cc0223: function declared implicitly") or as annotations in generated assembly files.

5.1.1.2 Whether each non-empty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3

Non-empty sequences of white-space characters are retained in translation phase 3.

J3.2 Environment

5.1.1.2 The mapping between physical source file multi-byte characters and the source character set in translation phase 1

When a multi-byte character is encountered, the compiler will interpret the constituent bytes as ASCII characters irrespective of what was intended by the author.

5.1.2.1 The name and type of the function called at program startup in a freestanding environment

The name of the function called at program startup is:

```
int main();
```

or, alternatively:

```
int main(int argc, char *argv[]);
```

5.1.2.1 The effect of program termination in a freestanding environment

On program termination, functions registered by the `atexit` function are called in reverse order of registration and then the processor is placed in an `IDLE` state.

5.1.2.2.1 An alternative manner in which the main function may be defined

---

CCES 2.9.0 C/C++ Compiler and Library Manual for Blackfin Processors

The default startup code source, which calls 'main', is provided and can be configured by the user.

Alternatively, startup code can be generated in the project settings within the IDE.

5.1.2.2.1 The values given to the strings pointed to by the argv argument to main

By default, argv[0] is a NULL pointer.

The values given to the strings pointed to by the argv argument can be defined by the user. For more information, see Support for argv/argc.

5.1.2.3 What constitutes an interactive device

An interactive device is considered a paired display screen and keyboard.

7.14 The set of signals, their semantics, and their default handling

The following signals are supported:

- SIGTERM
- SIGABRT
- SIGFPE
- SIGILL
- SIGINT
- SIGSEGV

7.14 After the handler is invoked, the disposition of the signal is not reset to SIG_DFL

By default, these signals are ignored.

7.14.1.1 Signal values other than SIGFPE, SIGILL, and SIGSEGV that correspond to a computational exception

There are no other signal values that correspond to a computational exception.

7.14.1.1 Signals for which the equivalent of signal(sig, SIG_IGN); is executed at program startup

- SIGTERM
- SIGABRT
- SIGFPE
- SIGILL
- SIGINT
- SIGSEGV

7.20.4.5 The set of environment names and the method for altering the environment list used by the getenv function

There is no default operating system and getenv will always return NULL.

7.20.4.6 The manner of execution of the string by the system function

The `system` function always returns 0.

J3.3 Identifiers

6.4.2 Which additional multi-byte characters may appear in identifiers and their correspondence to universal character names

Multi-byte characters may not be used in identifiers.

5.2.4.1, 6.4.2 The number of significant initial characters in an identifier

The maximum number of significant initial characters in an identifier is 15,000.

J3.4 Characters

The number of bits in a byte

8 bits.

5.2.1 The values of the members of the execution character set

The values of the execution character set are shown in the *Execution Character Set* table (with unprintable characters left blank).

Table 2-54: Execution Character Set

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x0																
0x1																
0x2	(space)	!	"	#	$	%	&	`	(	)	*	+	,	-	.	/
0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7	p	q	r	s	t	u	v	w	x	y	z	{			}	~	(DEL)

5.2.2 The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences

These values are shown in the *Escape Sequences in Execution Character Set* table.

Table 2-55: Escape Sequences in Execution Character Set

Escape	Value
\a	0x7

Table 2-55: Escape Sequences in Execution Character Set (Continued)

Escape	*Value*
\b	0x8
\f	0xC
\n	0xA
\r	0xD
\t	0x9
\v	0xB

6.2.5 The value of a char object into which has been stored any character other than a member of the basic execution character set

The resulting value is an integer which is derived from promoting an 8-bit character to type int. This value is always positive if the -unsigned-char switch (-unsigned-char) is used, but may be negative otherwise.

6.2.5, 6.3.1.1 Which of signed char or unsigned char has the same range, representation and behavior as "plain" char

A "plain" char has the same range and value as a signed char, except when the -unsigned-char switch (-unsigned-char) is used.

6.4.4.4, 5.1.1.2 The mapping of members of the source character set (in character constants and string literals) to members of the execution character set

Characters in the source file are interpreted as ASCII values, which are the same values used in the execution environment.

6.4.4.4 The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character

An integer character constant may contain up to 4 characters. If the constant contains between 2 and 4 characters, warning cc1994 will be issued. Using more than 4 characters will result in warning cc2226 being issued and all but the last 4 characters being discarded. No escape characters other than those specified in the C99 standard are supported, and these all map to a single byte in the execution environment.

6.4.4.4 The value of a wide character constant containing more than one multi-byte character, or containing a multi-byte character or escape sequence not represented in the extended execution character set

Where a wide character contains more than one multi-byte character, only the first character is retained and warning cc0026 will be issued. Subsequent characters are discarded. No escape characters other than those specified in the C99 standard are supported, and these all map to a single byte in the execution environment.

6.4.4.4 The current locale used to convert a wide character constant consisting of a single multi-byte character that maps to a member of the extended execution character set into a corresponding wide character code

Only the "C" locale is supported in Analog Devices tool chain and processors.

6.4.5 The current locale used to convert a wide string literal into corresponding wide character codes

Only the "C" locale is supported in Analog Devices tool chain and processors.

6.4.5 The value of a string literal containing a multi-byte character or escape sequence not represented in the execution character set

There are no escape sequences outside the basic or extended character sets.

J3.5 Integers

6.2.5 Any extended integer types that exist in the implementation

None

6.2.6.2 Whether signed integer types are represented using sign and magnitude, two's complement, or one's complement, and whether the extraordinary value is a trap representation or an ordinary value

Two's Complement:

- The sign bit being 1 and all value bits being zero is considered a normal number.

6.3.1.1 The rank of any extended integer type relative to another extended integer type with the same precision

N/A.

6.3.1.3 The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type

The hexadecimal value is copied and then interpreted as signed. e.g. `MAX_UINT` becomes -1.

6.5 The results of some bitwise operations on signed integers

Right shifts will retain the sign bit on a signed integer. All other bitwise operations treat signed integers as unsigned.

J3.6 Floating-Point

5.2.4.2.2 The accuracy of the floating-point operations and of the library functions in the <math.h> and <complex.h> that return floating-point results

This is a conforming freestanding implementation of C99. The accuracy of the library functions in these headers are therefore undocumented.

5.2.4.2.2 The rounding behaviors characterized by non-standard values of FLT_ROUNDS

`FLT_ROUNDS` is a standard value.

5.2.4.2.2 The evaluation methods characterized by non-standard negative values of FLT_EVAL_METHOD

`FLT_EVAL_METHOD` is undefined.

6.3.1.4 The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value

Round to nearest, ties to even.

6.3.1.5 The direction of rounding when a floating-point number is converted to a narrower floating-point number

Round to nearest, ties to even.

6.4.4.2 How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants

`FLT_RADIX` is defined as 2 in `<float.h>`, so floating-point constants are represented using standards-conforming rounding.

6.5 Whether and how floating expressions are contracted when not disallowed by the FP_CONTRACT pragma

This is a conforming freestanding implementation of C99. The `FP_CONTRACT` pragma is therefore not supported.

7.6.1 The default the state for the FENV_ACCESS pragma

This is a conforming freestanding implementation of C99, and the `FENV_ACCESS` pragma is only used for accessing the floating-point environment `fenv.h` - a header not required for such an implementation. As such this pragma is not supported.

7.6, 7.12 Additional floating-point exceptions, rounding modes, environments and classification, and their macro names

There are no additional floating-point exceptions, rounding modes, environments or classifications.

7.12.2 The default state for the FP_CONTRACT pragma

This is a conforming freestanding implementation of C99. The `FP_CONTRACT` pragma is therefore not supported.

F.9 Whether the "inexact" floating-point exception can be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation

The "inexact" floating-point exception is not supported for Blackfin processors.

F.9 Whether the "underflow" (and "inexact") floating-point exception can be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation

Floating-point exceptions are not supported on Blackfin processors.

ISO/IEC 14822:2003 C++ Standard (C++ Mode)

The subsection of this section refer to parts of the ISO/IEC 14822:2003 C++ Standard which have implementation-defined aspects.

1.7 The C++ Memory Model

The fundamental storage unit in the C + + memory model is the byte. A byte is at least large enough to contain any member of the basic execution character set and is composed of a contiguous sequence of bits, the number of which is implementation-defined.

8 bits.

1.9 Program Execution

What constitutes an interactive device is implementation-defined.

An interactive device is considered a paired display screen and keyboard.

2.1 Phases of Translation

Physical source file characters are mapped, in an implementation-defined manner, to the basic source character set (introducing new-line characters for end-of-line indicators) if necessary.

Characters in the source file are interpreted as ASCII values, which are also used in the execution environment.

Whether each non-empty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined.

Non-empty sequences of white-space characters are retained.

It is implementation-defined whether the source of the translation units containing these definitions is required to be available.

The source of the translation units containing these definitions must be available.

2.2 Character Sets

The values of the members of the execution character sets are implementation-defined, and any additional members are locale-specific.

The values of the execution character set are shown in the ***Execution Character Set for C++ Mode*** table (with un-printable characters left blank).

Table 2-56: The Execution Character Set for C++ Mode

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x0																
0x1																
0x2	(space)	!	"	#	$	%	&	`	(	)	*	+	,	-	.	/
0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	(DEL)

2.13.2 Character Literals

A multi-character literal has type int and implementation-defined value.

An integer character constant may contain up to 4 characters. If the constant contains between 2 and 4 characters, warning cc1994 will be issued. Using more than 4 characters will result in error cc0026 being issued.

The value of a wide-character literal containing multiple c-chars is implementation-defined.

Where a wide character contains more than one multi-byte character, only the first character is retained and warning cc0026 will be issued. Subsequent characters are discarded.

The value of a character literal is implementation-defined if it falls outside of the implementation-defined range defined for char (for ordinary literals) or wchar_t (for wide literals).

The least significant 8 bits are retained; all other bits are discarded.

2.13.4 String Literals

Whether all string literals are distinct (that is, are stored in non-overlapping objects) is implementation-defined.

Identical string literals within the same object file will not be distinct. That is, only one copy of the string will exist.

3.6.1 Main Function

An implementation shall not predefine the main function. This function shall not be overloaded. It shall have a return type of type int, but otherwise its type is implementation-defined.

The name of the function called at program startup is:

```
int main();
```

or, alternatively:

```
int main(int argc, char *argv[]);
```

The linkage (3.5) of main is implementation-defined.

main has external "C" linkage.

3.6.2 Initialization of Non-Local Objects

It is implementation-defined whether or not the dynamic initialization (8.5, 9.4, 12.1, 12.6.1) of an object of namespace scope is done before the first statement of main.

Dynamic initialization of an object of namespace scope is done before the first statement of main.

3.9 Types

For POD types, the value representation is a set of bits in the object representation that determines a value, which is one discrete element of an implementation-defined set of values.

All POD types are represented in the same format as in C.

The alignment of a complete object type is an implementation-defined integer value representing a number of bytes; an object is allocated at an address that meets the alignment requirements of its object type.

Compound types (structs, classes) are aligned on the boundary that matches the alignment of the most strictly-aligned member of the type.

Top-level arrays are always aligned on a word boundary, regardless of the underlying type. Arrays within structures are not aligned beyond the required alignment for their type.

3.9.1 Fundamental Types

It is implementation-defined whether a char object can hold negative values.

A `char` can hold negative values.

The value representation of floating-point types is implementation-defined.

The representations of the floating-point types are as follows:

- `float`
  - 32 bits (1 sign bit, 8 exponent bits, 32 mantissa bits)

    -3.4028234663852886E+38 to 3.4028234663852886E+38
- `double` (default setting)
  - 32 bits (1 sign bit, 8 exponent bits, 32 mantissa bits)

    -3.4028234663852886E+38 to 3.4028234663852886E+38
- `double` (when compiling with `"-double-size-64"`)
  - 64 bits (1 sign bit, 11 exponent bits, 52 mantissa bits)

    -1.797693134862315708e+308 to 1.797693134862315708e+308
- `long double`
  - 64 bits (1 sign bit, 11 exponent bits, 52 mantissa bits)

    -1.797693134862315708e+308 to 1.797693134862315708e+308

3.9.2 Compound Types

The value representation of pointer types is implementation-defined.

Pointer types are represented as 32-bit unsigned integers.

4.7 Integral Conversions

If the destination type is signed, the value is unchanged if it can be represented in the destination type (and bit-field width); otherwise, the value is implementation-defined.

When converting a signed integer to a smaller signed integer, the lower bits of the signed integer (of the size of the smaller signed integer) are copied to the smaller signed integer. If the topmost copied bit is set, this will result in a negative number.

4.8 Floating-Point Conversions

If the source value is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of those values.

Round to nearest, ties to even.

4.9 Floating-Integral Conversions

An rvalue of an integer type or of an enumeration type can be converted to an rvalue of a floating-point type. The result is exact if possible. Otherwise, it is an implementation-defined choice of either the next lower or higher representable value.

Round to nearest, ties to even.

5.2.8 Type Identification

The result of a typeid expression is an lvalue of static type const std::type_info (18.5.1) and dynamic type const std::type_info or const name where name is an implementation-defined class derived from std::type_info which preserves the behavior described in 18.5.1.

The result of a `typeid` expression is an lvalue of static type `const std::type_info` and dynamic type `const std::type_info`.

5.2.10 Reinterpret Cast

The mapping performed by reinterpret_cast is implementation-defined.

For an expression "`reinterpret_castT>(v)`", the bits in the object representation of `v` will be treated as type as an object of type "`T`".

A pointer can be explicitly converted to any integral type large enough to hold it. The mapping function is implementation-defined.

The bit pattern of the pointer is interpreted as the integral type. No sign extension is performed if the integral type is larger than the pointer.

A value of integral type or enumeration type can be explicitly converted to a pointer. A pointer converted to an integer of sufficient size (if any such exists on the implementation) and back to the same pointer type will have its original value; mappings between pointers and integers are otherwise implementation-defined.

A cast from pointer to integer results in the most-significant bits being discarded if the size of the pointer is larger than the integer. If the pointer is smaller than the integer type being cast to, the integer will be zero-extended.

A cast from integer to pointer results in the most-significant bits being discarded if the size of the integer is larger than the pointer. If the integer is smaller than the pointer type being cast to, the pointer will be sign-extended.

5.3.3 Sizeof

sizeof(char), sizeof(signed char) and sizeof(unsigned char) are 1; the result of sizeof applied to any other fundamental type (3.9.1) is implementation-defined. [Note: in particular, sizeof(bool) and sizeof(wchar_t) are implementation-defined.]

Sizes are as shown in the *Sizes of C++ Standard Types* table.

Table 2-57: Sizes of C++ Standard Types

char (signed, unsigned)	1
short (signed, unsigned)	2
int (signed, unsigned)	4
long (signed, unsigned)	4
long long (signed, unsigned)	8
float	4
double (default)	4
double (-double-size-64)	8
long double	8
bool	1
wchar_t	4

5.6 Multiplicative Operators

The binary / operator yields the quotient, and the binary % operator yields the remainder from the division of the first expression by the second. If the second operand of / or % is zero the behavior is undefined; otherwise (a/b)*b + a %b is equal to a. If both operands are nonnegative then the remainder is nonnegative; if not, the sign of the remainder is implementation-defined.

If the first operand is negative, the sign of the remainder is negative, otherwise the sign of the remainder is nonnegative.

5.7 Additive Operators

When two pointers to elements of the same array object are subtracted, the result is the difference of the subscripts of the two array elements. The type of the result is an implementation-defined signed integral type; this type shall be the same type that is defined as ptrdiff_t in the <cstddef> header (18.1).

The type of the result is `long int`.

5.8 Shift Operators

The value of E1 >> E2 is E1 right-shifted E2 bit positions. If E1 has an unsigned type or if E1 has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of E1 divided by the quantity 2 raised to the power E2. If E1 has a signed type and a negative value, the resulting value is implementation-defined.

Right shifts will retain the sign bit on a signed integer.

7.1.5.2 Simply Type Specifiers

It is implementation-defined whether bit-fields and objects of char type are represented as signed or unsigned quantities.

By default, bit-fields and objects of `char` type are represented as signed quantities.

Bit-fields can represented as unsigned quantities by using the compiler switch `-unsigned-bitfield` (-unsigned-bitfield).

`char`s can represented as unsigned quantities by using the compiler switch `-unsigned-char` (-unsigned-char).

7.2 Enumeration Declarations

It is implementation-defined which integral type is used as the underlying type for an enumeration except that the underlying type shall not be larger than int unless the value of an enumerator cannot fit in an int or unsigned int.

The underlying type for an enumeration shall be `int`.

7.4 The asm Declaration

The meaning of an asm declaration is implementation-defined.

For more information, see Inline Assembly Language Support Keyword (asm).

7.5 Linkage Specifications

The string-literal indicates the required language linkage. The meaning of the string-literal is implementation-defined.

Three string-literals are supported:

- `"C"`- the function name in the source file is prefixed with an underscore (`"_"`) in the object file.
- `"C++"` - the function name is mangled according to the compiler's name mangling rules.
- `"asm"` - the function name in the source file is used in the object file without a prefix or name-mangling.

Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages is implementation-defined and language-dependent.

Three string-literals are supported:

- `"C"` - the function name in the source file is prefixed with an underscore (`"_"`) in the object file.
- `"C++"` - the function name is mangled according to the compiler's name mangling rules.
- `"asm"` - the function name in the source file is used in the object file without a prefix or name-mangling.

9.6 Bit-Fields

Allocation of bit-fields within a class object is implementation-defined. Alignment of bit-fields is implementation-defined.

Bit-fields are stored using a little-endian representation.

Bit-fields are aligned such that they do not cross a 32-bit word boundary (for bit-fields of type `char`, `short`, `int` or `long`) or a 64-bit boundary (for bit-fields of type `long long`). For example, a 24-bit bit-field can be placed immediately after an 8-bit bit-field, but a 25-bit bitfield member will be aligned on the next 32-bit boundary.

It is implementation-defined whether a plain (neither explicitly signed nor unsigned) char, short, int or long bit-field is signed or unsigned.

Plain bit-fields are signed.

14 Templates

A template name has linkage (3.5). A non-member function template can have internal linkage; any other template name shall have external linkage. Entities generated from a template with internal linkage are distinct from all entities generated in other translation units. A template, a template explicit specialization (14.7.3), or a class template partial specialization shall not have C linkage. If the linkage of one of these is something other than C or C++, the behavior is implementation-defined.

Only C++ linkage is supported for templates.

14.7.1 Implicit Instantiation

There is an implementation-defined quantity that specifies the limit on the total depth of recursive instantiations, which could involve more than one template.

The limit on the total depth of recursive instantiations is 64.

15.5.1 The terminate() Function

In the situation where no matching handler is found, it is implementation-defined whether or not the stack is unwound before terminate() is called.

The stack is not unwound before the call to `terminate()`.

15.5.2 The unexpected() Function

If the exception-specification does not include the class std::bad_exception (18.6.2.1) then the function terminate() is called, otherwise the thrown exception is replaced by an implementation-defined object of the type std::bad_exception and the search for another handler will continue at the call of the function whose exception-specification was violated.

The object of the type `std::bad_exception` contains the string "bad exception".

16.1 Conditional Inclusion

Whether the numeric value for these character literals matches the value obtained when an identical character literal occurs in an expression (other than within a #if or #elif directive) is implementation-defined.

The numeric value for these character literals matches the value obtained when an identical character literal occurs in an expression.

Also, whether a single-character character literal may have a negative value is implementation-defined.

A single-character may have a negative value.

16.2 Source File Inclusion

Searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the < and > delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

Include files, whose names are not absolute path names and that are enclosed in "..." when included, are searched for in the following directories in this order:

- The directory containing the current input file (the primary source file or the file containing the #include).
- Any directories specified with the -I switch (-I `directory[{,|;} directory...]`) in the order they are listed on the command line.
- Any directories on the standard list: `<install_path>\...\include`.

The mapping between the delimited sequence and the external source file name is implementation-defined.

The source file character sequence is mapped to its corresponding ASCII character sequence.

A #include preprocessing directive may appear in a source file that has been read because of a #include directive in another file, up to an implementation-defined nesting limit.

The compiler does not define a nesting limit for #include directives.

16.6 Pragma Directive

A preprocessing directive #pragma causes the implementation to behave in an implementation-defined manner.

For more information, see Pragmas.

16.8 Predefined Macro Names

If the date of translation is not available, an implementation-defined valid date is supplied.

The macro __DATE__ will be defined as "[date unknown]".

If the time of translation is not available, an implementation-defined valid time is supplied.

The macros __TIME__ will be defined as "[time unknown]".

Whether __STDC__ is predefined and if so, what its value is, are implementation-defined.

`__STDC__` is predefined with the value 1.

17.4.4.5 Reentrancy

Which of the functions in the C++ Standard Library are not reentrant subroutines is implementation-defined.

The following functions are not reentrant in the C++ Standard library, as implemented in CCES:

- Functions that use streams.

- Dynamic memory allocation functions (`new`, `delete`, etc.).

- The exceptions handling support routines.

Although these functions are not reentrant, thread-safe versions of them are implemented in the multi-threaded C++ library. For more information, see Library Function Re-Entrancy and Thread Safety in the *C/C++ Run-Time Library* chapter.

17.4.4.8 Restrictions on Exception Handling

Any other functions defined in the C++ Standard Library that do not have an exception-specification may throw implementation-defined exceptions unless otherwise specified.

The *Functions Which Throw Exceptions* table shows which functions may throw the following exceptions, if the application is built with exceptions enabled.

Table 2-58: Functions Which Throw Exceptions

Function	Exception Type
ios_base::clear	failure
locale::locale	runtime_error
_Locinfo::_Addcats	runtime_error
_String_base::_Xlen	length_error
_String_base::_Xran	out_of_range
array new and delete operators	bad_alloc

18.3 Start and Termination

Exit() - Finally, control is returned to the host environment. If status is zero or EXIT_SUCCESS, an implementation-defined form of the status successful termination is returned. If status is EXIT_FAILURE, an implementation-defined form of the status unsuccessful termination is returned. Otherwise the status returned is implementation-defined.

The status is written to the variable `_exit_value` and the program will idle at the label `__lib_prog_term`. R0 (the return register) always is set to zero.

18.4.2.1 Class bad_alloc

The result of calling what() on the newly constructed object is implementation-defined.

`what()` returns the string "bad allocation".

virtual const char* what() const throw(); Returns: An implementation-defined NTBS

`what()` returns the string "bad allocation".

18.5.1 Class type_info

const char* name() const; Returns: an implementation-defined NTBS

The *Strings Returned by Name()* table shows the string returned by the name() function for the basic types.

Table 2-59: Strings Returned by Name()

Type	String
bool	b
char	c
signed char	a
unsigned char	h
(signed) short	s
unsigned short	t
(signed) int	i
unsigned int	j
(signed) long	l
unsigned long	m
(signed) long long	x
unsigned long long	y
float	f
double	d
long double	e
wchar_t	w

18.5.2 Class bad_cast

virtual const char* what() const throw(); Returns: An implementation-defined NTBS

Calling `what()` returns the string "bad cast".

18.5.3 Class bad_typeid

bad_typeid() throw(); Notes: The result of calling what() on the newly constructed object is implementation-defined.

Calling `what()` returns the string "bad typeid".

virtual const char* what() const throw(); Returns: An implementation-defined NTBS

Calling `what()` returns the string "bad typeid".

18.6.1 Class Exception

exception& operator=(const exception&) throw(); Notes: The effects of calling what() after assignment are implementation-defined.

Calling `what()` returns the string "unknown".

virtual const char* what() const throw(); Returns: An implementation-defined NTBS

Calling `what()` returns the string "unknown".

18.6.2.1 Class bad_exception

bad_exception() throw(); Notes: The result of calling what()on the newly constructed object is implementation-defined.

Calling `what()` returns the string "bad exception".

virtual const char* what() const throw(); Returns: An implementation-defined NTBS

Calling `what()` returns the string "bad exception".

21 Strings Library

The type streampos is an implementation-defined type that satisfies the requirements for POS_T in 21.1.2.

`streampos` is a `typedef` of the `fpos` class.

The type streamoff is an implementation-defined type that satisfies the requirements for OFF_T in 21.1.2.

`streamoff` is a `typedef` of the `long` type.

The type mbstate_t is defined in <cwchar> and can represent any of the conversion states possible to occur in an implementation-defined set of supported multi-byte character encoding rules.

Multi-byte characters are not supported in Analog Devices Compiler, so no multi-byte characters may be used in identifiers.

21.1.3.2 struct <char_traitswchar_t>

The type wstreampos is an implementation-defined type that satisfies the requirements for POS_T in 21.1.2.

The type `wstreampos` not supported in Analog Devices toolset.

The type mbstate_t is defined in <cwchar> and can represent any of the conversion states possible to occur in an implementation-defined set of supported multi-byte character encoding rules.

Multi-byte characters are not supported in Analog Devices Compiler, so no multi-byte characters may be used in identifiers.

22.1.1.3 Locale Members

*basic_string<char> name() const; Returns: The name of *this, if it has one; otherwise, the string "*". If *this has a name, then locale(name().c_str()) is equivalent to *this. Details of the contents of the resulting string are otherwise implementation-defined.*

name returns the name of *this, if it has one; otherwise, the string "*".

22.2.1.3 ctype Specializations

The implementation-defined value of member table_size is at least 256.

The value of member table_size is 256.

22.2.1.3.2 ctype<char> Members

In the following member descriptions, for unsigned char values v where (v >= table_size), table()[v] is assumed to have an implementation-defined value (possibly different for each such value v) without performing the array look-up.

As table_size has the value 256, it is not possible for v to be greater than or equal to table_size.

22.2.5.1.2 time_get Virtual Functions

iter_type do_get_year(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm t) const; Effects: Reads characters starting at s until it has extracted an unambiguous year identifier. It is implementation-defined whether two-digit year numbers are accepted, and (if so) what century they are assumed to lie in. Sets the t->tm_yearmember accordingly.*

If the two-digit year is less than '69', it is assumed that the year is in the 21st century (i.e. 2000 -> 2068); otherwise, it is assumed that the year is in the 20th century.

22.2.5.3.2 time_put Virtual Functions

Effects: Formats the contents of the parameter t into characters placed on the output sequences. Formatting is controlled by the parameters format and modifier, interpreted identically as the format specifiers in the string argument to the standard library function strftime(). except that the sequence of characters produced for those specifiers that are described as depending on the C locale are instead implementation-defined.

The *Outputs for time_put Specifiers* table shows the character sequences produced for each specifier that depends on the C locale.

Table 2-60: Outputs for time_put Specifiers

Specifier	*Characters*
%a	"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"
%A	"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"
%b	"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"

Table 2-60: Outputs for time_put Specifiers (Continued)

Specifier	Characters
%B	"January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"
%c	Date and time in the format: DDD MMM DD HH:MM:SS YYYY For example, "Sat Jan 31 23:59:59 2011".
%p	"AM", "PM"
%x	Date in the format: MM/DD/YY For example, "12/31/12".
%X	Time in the format: HH:MM:SS For example, "23:59:59".

22.2.7.1.2 Messages Virtual Functions

catalog do_open(const basic_string<char>& name, const locale& loc) const; Returns: A value that may be passed to get()to retrieve a message, from the message catalog identified by the string name according to an implementation-defined mapping. The result can be used until it is passed to close().

This function has no effect.

string_type do_get(catalog cat, int set, int msgid, const string_type& dfault) const; Returns: A message identified by arguments set, msgid, and dfault, according to an implementation-defined mapping

The function `do_get` always returns the string pointed to by `dfault`.

void do_close(catalog cat) const; Notes: The limit on such resources, if any, is implementation-defined.

This function has no effect.

26.2.8 Complex Transcendentals

The value returned for pow(0,0) is implementation-defined.

This is a conforming freestanding implementation of C++. Complex transcendentals are not supported.

27.1.2 Positioning Type Limitations

The classes of clause 27 with template arguments charT and traits behave as described if traits::pos_type and traits::off_type are streampos and streamoff respectively. Except as noted explicitly below, their behavior when traits::pos_type and traits::off_type are other types is implementation-defined.

`traits::pos_type` and `traits::off_type` are `streampos` and `streamoff` respectively.

27.4.1 Types

The type streamoff is an implementation-defined type that satisfies the requirements of 27.4.3.2.

`streamoff` is of type `long`.

27.4.2.4 ios_base Static Members

bool sync_with_stdio(bool sync = true); Effects: If any input or output operation has occurred using the standard streams prior to the call, the effect is implementation-defined.

`iostream` objects are always synchronized with the standard streams. This function has no effect.

27.4.4.3 basic_ios iostate Flags Functions

If (rdstate() & exceptions()) == 0, returns. Otherwise, the function throws an object fail of class basic_ios::failure (27.4.2.1.1), constructed with implementation-defined argument values.

If `'ios_base::badbit'` is set, the exception is created with the string "ios_base::badbit set".

If `'ios_base::failbit'` is set, the exception is created with the string "ios_base::failbit set".

27.7.1.3 Overridden Virtual Functions

basic_streambuf<charT,traits> setbuf(charT* s, streamsize n); Effects: implementation-defined, except that setbuf(0,0)has no effect*

`streambuf()` has no effect.

27.8.1.4 Overridden Virtual Functions

basic_streambuf setbuf(char_type* s, streamsize n); Effects: If setbuf(0,0) is called on a stream before any I/O has occurred on that stream, the stream becomes unbuffered. Otherwise the results are implementation-defined.*

If `setbuf(s, n)` is called before any I/O has occurred, the buffer 's', of size 'n', is used by the I/O routines. Calls to `setbuf()` on a stream after I/O has occurred are ignored.

int sync(); Effects: If a put area exists, calls filebuf::overflow to write the characters to the file. If a get area exists, the effect is implementation-defined.

The `sync()` function has no effect on the get area.

C.2.2.3 Macro NULL

The macro NULL, defined in any of <clocale>, <cstddef>, <cstdio>, <cstdlib>, <cstring>, <ctime>, or <cwchar>, is an implementation-defined C++ null pointer constant in this International Standard (18.1).

The macro `NULL` is defined as 0.

D.6 Old iostreams Members

The type streamoff is an implementation-defined type that satisfies the requirements of type OFF_T (27.4.1).

`streamoff` is a `typedef` of the `'long'` type.

The type streampos is an implementation-defined type that satisfies the requirements of type POS_T (27.2).

`streampos` is a `typedef` of the `fpos` class.

3   Optimal Performance from C/C++ Source Code

This chapter provides guidance on tuning your application to achieve the best possible code from the compiler. Since implementation choices are available when coding an algorithm, understanding their impact is crucial to attaining optimal performance.

This chapter contains:

- General Guidelines provides a four-step basic strategy for designing applications. It also describes topics such as data types, memory usage, and indexed arrays versus pointers.

- Improving Conditional Code describes the `expected_true` and `expected_false` built-in functions, which control the optimization of conditional branches.

- Loop Guidelines describes how to help the compiler produce the most efficient loop code, including keeping loops short, and avoiding unrolling loops and loop-carried dependencies.

- Manipulating Fixed-Point and Fractional Data discusses ways to manipulate fixed-point and fractional data.

- Using Built-In Functions in Code Optimization describes how to use built-in functions to use efficiently low-level features of the processor hardware while programming in C.

- Smaller Applications: Optimizing for Code Size provides tips and techniques for optimizing the application to achieve good performance while meeting code space constraints.

- Using Pragmas for Optimization describes how to use pragmas to fine-tune source code.

- Optimization Switches lists compiler switches useful during the optimization process.

- How Loop Optimization Works introduces concepts used in loop optimization.

- Assembly Optimizer Annotations describes annotations, which indicate how close to optimal a program is, and suggest what else can be done to improve the generated code.

- Analyzing Your Application describes various techniques that can be used to analyze and debug a program. Instrumented profiling, code coverage, and stack and heap tracing are discussed.

This chapter helps you get maximal code performance from the compiler. Most of these guidelines also apply when optimizing for minimum code size, although some techniques specific to that goal are also discussed.

The first section looks at some general principles, and explains how the compiler can help your optimization effort. Optimal coding styles are then considered in detail. Special features such as compiler switches, built-in functions, and pragmas are also discussed. The chapter includes a short example to demonstrate how the optimizer works.

Small examples are included throughout this chapter to demonstrate points being made. Some show recommended coding styles, while others identify styles to be avoided or code that it can be possible to improve. These styles are commented in the code as GOOD and BAD, respectively.

General Guidelines

This section contains:

- How the Compiler Can Help
- The volatile Type Qualifier
- Data Types
- Getting the Most From IPA
- Indexed Arrays Versus Pointers
- Using Function Inlining
- Using Inline asm Statements
- Memory Usage

Remember the following strategy when writing an application:

1. Choose the language as appropriate. Your first decision is whether to implement your application in C or C++. Performance considerations may influence this decision. C++ code using only C features has very similar performance to pure C code. Many higher level C++ features (for example, those resolved at compilation, such as namespaces, overloaded functions and also inheritance) have no performance cost.

   However, use of some other features may degrade performance. Carefully weigh performance loss against the richness of expression available in C++ (such as virtual functions or classes used to implement basic data types).

2. Choose an algorithm suited to the architecture being targeted. For example, the target architecture will influence any trade-off between memory usage and algorithm complexity.

3. Code the algorithm in a simple, high-level generic form. Keep the target in mind, especially when choosing data types.

4. Tune critical code sections. After your application is complete, identify the most critical sections. Carefully consider the strengths of the target processor and make non-portable changes where necessary to improve performance.

How the Compiler Can Help

The compiler provides many facilities to help the programmer to achieve optimal performance, including the compiler optimizer, statistical profiler, profile-guided optimizer (PGO), and interprocedural optimizers.

This section contains:

- Using the Compiler Optimizer

- Using Compiler Diagnostics

- Using Profile-Guided Optimization

- Using Interprocedural Optimization

Using the Compiler Optimizer

There is a vast difference in performance between code compiled optimized and code compiled non-optimized. In some cases, optimized code can run ten or twenty times faster. Always use optimization when measuring performance or shipping code as product.

The optimizer in the C/C++ compiler is designed to generate efficient code from source that has been written in a straightforward manner. The basic strategy for tuning a program is to present the algorithm in a way that gives the optimizer the best possible visibility of the operations and data, and hence the greatest freedom to safely manipulate the code. Future releases of the compiler will continue to enhance the optimizer. Expressing algorithms simply will provide the best chance of benefiting from such enhancements.

The default setting ("Debug" configuration within the IDE) is for non-optimized compilation in order to assist programmers in diagnosing problems with their initial coding. The optimizer is enabled in the IDE by selecting *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > General > Enable optimization*, or by using the -O[0|1] switch. A "release" build from within CCES automatically enables optimization.

Using Compiler Diagnostics

There are many features of the C and C++ languages that, while legal, often indicate programming errors. There are also aspects that are valid but may be relatively expensive for an embedded environment. The compiler can provide diagnostics which save time and effort in characterizing source-related problems.

These diagnostics are particularly important for obtaining high-performance code, since the optimizer aggressively transforms the application to yield the best performance, discarding unused or redundant code. If this code is redundant because of a programming error (such as omitting an essential `The volatile Type Qualifier` qualifier from a declaration), then the code will behave differently from a non-optimized version. Using the compiler's diagnostics may help you identify such situations before they become problems.

The diagnostic facilities are described in the following sections:

- Warnings, Annotations and Remarks

- Run-Time Diagnostics

- Steps for Developing Your Application

Warnings, Annotations and Remarks

By default, the compiler emits warnings to the standard error stream at compile-time when it detects a problem with the source code. Warnings can be disabled individually, with the `-Wsuppress` switch or as a class, with the `-w` switch, disabling all warnings and remarks. (See the Compiler chapter.) However, disabling warnings is inadvisable until each instance has been investigated for problems.

A typical warning would be: a variable being used before its value has been set.

Remarks are diagnostics that are less severe than warnings. Like warnings, they are produced at compile-time to the standard error stream, but unlike warnings, remarks are suppressed by default. Remarks are typically for situations that are probably correct, but less than ideal. Remarks may be enabled as a class with the `-Wremarks` switch (see the Compiler chapter), or by choosing Project > Properties > C/C++ Build > *Settings > Tool Settings > Compiler > Warning > Warning/annotation/remark control* to *Errors, Warnings, Annotations and Remarks* in the IDE.

A typical remark would be: a variable being declared, but never used.

A remark may be promoted to a warning through the `-Wwarn` switch. Remarks and warnings may be promoted to errors through the `-Werror`, -w switch.

Annotations are diagnostics that are between warnings and remarks in severity. Like remarks, annotations are usually suppressed. Where remarks comment on the input source file, annotations provide information about the code the compiler has generated from the source file.

A typical annotation would be: using a volatile variable within a loop limits optimization.

Both annotations and remarks can be viewed in the IDE; they are listed as "infos" in the Problems view, and an "information" icon appears in the gutter of the source file's view, adjacent to the associated line. Hovering over the gutter icon displays the annotations and remarks for the line.

Annotations are also emitted to the generated assembly file, as comments. For more information, see Assembly Optimizer Annotations.

Run-Time Diagnostics

Although the compiler can identify many potential problems through its static analysis, some problems only become apparent at run-time. The compiler and libraries provide a number of facilities for assisting in identifying such problems. These facilities are:

- Run-time diagnostics, where the compiler plants additional code to check for common programming errors. For more information, see Run-Time Checking in the Compiler chapter.

- Stack overflow detection, where the compiler ensures that the stack does not run out of space. For more information, see Stack Overflow Detection.

- Heap debugging, where the compiler links the application with an enhanced version of the heap library, to detect memory leaks and other common dynamic-memory issues. For more information, see Heap Debugging.

Steps for Developing Your Application

To improve overall code quality:

1. Enable remarks and build the application. Gather all warnings and remarks generated.

2. Examine the generated diagnostics and choose those message types that you consider most important. For example, you might select just `cc0223`, a remark that identifies implicitly-declared functions.

3. Promote those remarks and warnings to errors, using the `-Werror` switch (for example, "`-Werror 0223`"), and rebuild the application. The compiler will now fault such cases as errors, so you will have to fix the source to address the issues before your application will build.

4. Once your application rebuilds, repeat the process for the next most important diagnostics.

5. When you have dealt with the diagnostics you consider significant, rebuild your application with run-time diagnostics enabled, and run your regression tests, to see whether any problems lurk. (Given the overheads of run-time diagnostics, you will probably find it better to only enable one form at a time.)

6. Once your application runs successfully with each form of run-time diagnostic, disable run-time diagnostics and rebuild your application for release.

Diagnostics you might typically consider first include:

- `cc0223`: function declared implicitly

- `cc0549`: variable used before its value is set

- `cc1665`: variable is possibly used before its value is set, in a loop

- `cc0187`: use of "=" where "==" may have been intended

- `cc1045`: missing return statement at the end of non-void function

- `cc0111`: statement is unreachable

If you have particular cases that are correct for your application, do not let them prevent your application from building because you have raised the diagnostic to an error. For such cases, temporarily lower the severity again within the source file in question by using `#pragma diag` (see Diagnostic Control Pragmas in the Compiler chapter).

Using Profile-Guided Optimization

Profile-guided optimization (PGO) is an excellent way to tune the compiler's optimization strategy for the typical run-time behavior of a program. There are many program characteristics that cannot be known statically at compile-time but can be provided through PGO. The compiler can use this knowledge to improve its code generation. The benefits include more accurate branch prediction, improved loop transformations, and reduced code size. The technique is most relevant where the behavior of the application over different data sets is expected to be very similar.

> **NOTE:** The data gathered during the profile-guided optimization process can also be used to generate a code coverage report. See Profile-Guided Optimization and Code Coverage.

Profile-guided optimization can be performed on applications running on both hardware and simulators. The functionality supported and the steps required are different in each case. A summary of these differences is listed in the *Differences Between Profile-Guided Optimization for Simulators and Hardware* table.

Table 3-1: Differences Between Profile-Guided Optimization for Simulators and Hardware

Profile-Guided Optimization for Simulators	*Profile-Guided Optimization for Hardware*
Is non-intrusive to the application. No code or data space needs to be reserved for the profiling.	Is intrusive. Profiling requires both code and data space to be reserved in the application.
Does not impact performance. Profiling is performed in the background by the simulator.	Impacts performance. Profiling is performed on the processor as part of the application.
Does not support multi-threaded applications.	Supports multi-threaded applications.
Can only profile application where peripherals are simulated by the simulator.	Run on hardware allowing the profiling of applications that use custom hardware.

Profile-guided optimization using the simulator is a non-intrusive process: the application code is not modified to gather the profiling data. Multi-threaded applications cannot be profiled using the simulator-based method of profile gathering.

Profile-guided optimization for applications running on hardware offers support for multi-threaded applications and applications that cannot be run on the simulator (for example, due to custom hardware or requiring input from peripherals not supported by the simulator). However, the hardware-based profiling method is more intrusive to the application, as it requires instruction and data memory.

Using Profile-Guided Optimization With a Simulator

The process of PGO execution with a simulator is illustrated in the ***PGO Process When Targeting a Simulator*** figure.
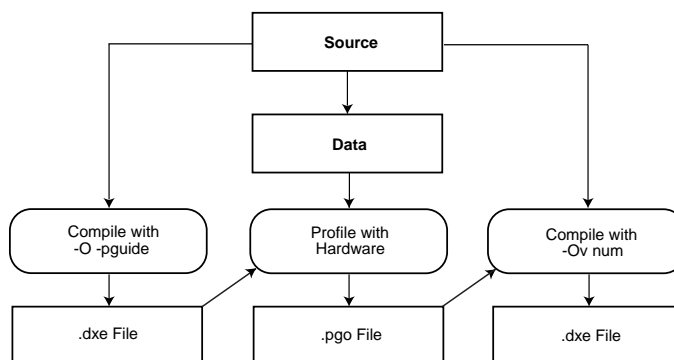
Figure 3-1: PGO Process When Targeting a Simulator

1. Compile the application with the `-pguide` switch or choose ***Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Profile-guided Optimization > Prepare application to create new profile*** option/`-pguide`. This creates an executable file containing the necessary instrumentation for gathering profile data. For best results, click ***General*** (under ***Compiler***) in the tree control and select ***Enable optimization***/`-O` switch or ***Interprocedural optimization***/-ipa switch.

2. Gather the profile. Run the executable under the simulator, with one or more training data sets.

    a. Load the application into the simulator.

b. Enable profiling, via *Target > PGO > Simulator > Start*.

c. Run the application, with the desired training set.

d. Save the profile, via *Target > PGO > Simulator > Stop and save*.

e. Repeat the process with the next training set.

The training data sets should be representative of the data that you expect the application to process in the field. Note that unrepresentative training data sets can cause performance degradations when the application is used on real data. The profile is stored in a file with the extension `.pgo`.

3. Recompile the application using this gathered profile data:

a. Turn off the `-pguide` switch or choose *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Profile-guided Optimization > Prepare application to create new profile*.

b. Place the .pgo file on the command line or include it in the list of profiles under *Optimize using existing profiles*.

c. Ensure optimization is enabled:

Click *General* (under *Compiler)* in the tree control and select *Enable optimization* /`-O` switch and/or *Interprocedural optimization*/`-ipa` switch.

NOTE: When C/C++ source files are specified in a compiler command line, any specified `.pgo` files will be used to guide compilation. However, any recompilation due to `.doj` files provided on the command line will reread the same `.pgo` file as when the source was previously compiled. For example, `prof2.pgo` is ignored in the following commands:

```
cc21k -O f2.c -o f2.doj prof1.pgo
cc21k -o prog.dxe f1.asm f2.doj prof2.pgo
```

For an example application that demonstrates how to use PGO, refer to Using PGO in Function Profiling.

Using Profile-Guided Optimization With Hardware

The process of PGO execution with hardware is illustrated in the *PGO Process on Hardware* figure.
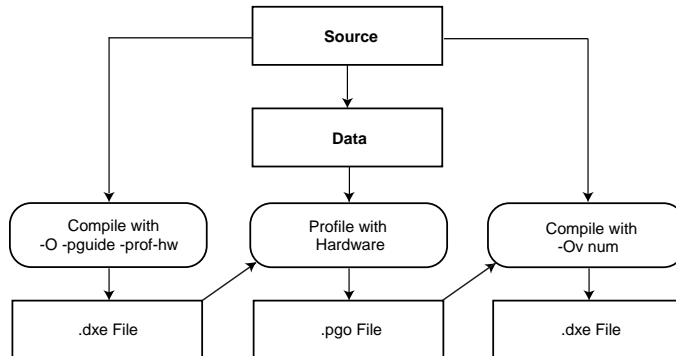
Figure 3-2: PGO Process on Hardware

1. Compile the application with the `-pguide` switch, which is equivalent to *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Profile-guided Optimization > Prepare application to create new profile*, and the `-prof-hw` switch, which is equivalent to the Project > Properties > C/C++ Build > *Settings > Tool Settings > Compiler > Profile-guided Optimization > Gather profile using hardware* option.

   This creates an executable file containing the necessary instrumentation for gathering profile data when run on hardware.

   For best results, click *General* (under *Compiler)* in the tree control and select *Enable optimization*/`-O` switch and/or *Interprocedural optimization*/-ipa switch.

   See the Compiler chapter for details.

2. Gather the profile. Run the executable on the hardware with one or more training data sets. These training data sets should be representative of the data that you expect the application to process in the field. Note that unrepresentative training data sets can cause performance degradations when the application is used on real data. The profile is stored in files with the extension `.pgo` and `.pgt`.

3. Recompile the application using this gathered profile data:

   a. Turn off the `-prof-hw` switch or choose *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Profile-guided Optimization > Gather profile using hardware*.

   b. Turn off the `-pguide` switch or choose *Prepare application to create new profile*.

   c. Place the .pgo file on the command line, or include it in the list of profiles under *Optimize using existing profiles*. The `.pgo` file contains a reference to the `.pgt` file, so this automatically includes the `.pgt` file.

4. Ensure optimization is enabled:

   Click *General* (under *Compiler)* in the tree control and select *Enable optimization* /`-O` switch and/or *Interprocedural optimization*/-ipa switch.

NOTE: PGO for hardware works by planting function calls into your application which record the execution count (and in multi-threaded cases, the thread identifier) at certain points. Applications built with PGO for hardware should be used for development and should not be released.

NOTE: PGO for hardware requires that an I/O device is available in the application to produce its profiling data. The default I/O device is used to perform I/O operations for PGO.

NOTE: PGO for hardware flushes any remaining profile data still pending when `exit()` is invoked. Multi-threaded applications may need to flush data explicitly.

NOTE: When C/C++ source files are specified in a compiler command line, any specified `.pgo` files will be used to guide compilation. However, any recompilation due to `.doj` files provided on the command line will reread the same `.pgo` file a s when the source was previously compiled.

For example, `prof2.pgo` is ignored in the following commands:

```
cc21k -O f2.c -o f2.doj prof1.pgo
cc21k -o prog.dxe f1.asm f2.doj prof2.pgo
```

Flushing PGO Data in Multi-Threaded and Non-Terminating Applications

Applications that are optimized with profile-guided optimization for hardware must ensure that the profiling information is flushed to the host machine. Flushing occurs when any of the following conditions are met:

- In an application linked with the single-threaded run-time libraries, data is flushed when the profile-guided optimization data buffer is full.

- In an application linked with the threadsafe run-time libraries, once the profile-guided optimization data buffer is 75% full, data will be flushed at the next available opportunity.

- When the profile-guided optimization maximum flush interval has been exceeded. By default, the maximum flush interval is 10 minutes.

- When the application explicitly requests a flush of the profile-guided optimization data.

Applications which do not terminate (and multi-threaded applications) must be modified to flush the data at an appropriate time. To request a flush of the data, add a call to the function `pgo_hw_request_flush()`. The example code in the *Flushing Profile-Guided Optimization Data from an Application* example shows a function that has been modified to flush the profile-guided data. The required changes are conditionally included when the preprocessor macro `_PGO_HW` is defined. The `_PGO_HW` macro is only defined when the application is compiled with the `-pguide` and `-prof-hw` compiler switches. Flushing the data to the host is a cycle-intensive operation, so you should consider carefully where to place the call to flush within your application. In the *Flushing Profile-Guided Optimization Data from an Application* example, the flush request has been placed in function `do_pgo_flush()`, which is called after the critical data loop in an attempt to reduce the impact of the profiling on the application's behavior. `do_pgo_flush()` is marked by `#pragma pgo_ignore`, so that no profile information is generated for the function. Isolating the flushing action in this manner is important because the verifies that a gathered profile matches the function's structure, before using the profile in optimization; if `pgo_hw_request_flush()` was conditionally called directly from `main_loop()`, when the application was recompiled with the gathered profile, but without the `-prof-hw` switch, the compiler would see that the call was now absent, making the profile invalid, and causing the optimizer to disregard the profile.

Flushing Profile-Guided Optimization Data from an Application

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

extern int get_task(void);

#pragma pgo_ignore
static void do_pgo_flush(void) {
   #if defined(_PGO_HW)
      pgo_hw_request_flush();
   #endif }

   void main_loop(void) {
      while ( 1 ) {
         int task = get_task();
         if ( task == 1 ) {
            // perform critical data loop
            do_pgo_flush();
         } else {
            // other tasks
         }
      }
   }
}
```

Profile-Guided Optimization and Multiple Source Uses

In some applications, it is convenient to build the same source file in more than one way within the same application. For example, a source file might be conditionally compiled with different macro settings. Alternatively, the same file might be compiled once, but linked more than once into the same application in a multi-core or multi-processor environment. In such circumstances, the typical behaviors of each instance in the application might differ. Identify and build the instances separately so that they can be profiled individually and optimized according to their typical use.

The -pgo-session switch (or the -pgo-session *session-id* option in the Compiler chapter) is used to separate profiles in such cases. It is used during both stage 1, where the compiler instruments the generated code for profiling, and during stage 3, where the compiler uses gathered profiles to guide the optimization.

During stage 1, when the compiler instruments the generated code, if the -pgo-session switch is used, then the compiler marks the instrumentation as belonging to the session's session-id.

During stage 3, when the compiler reads gathered profiles, if the -pgo-session switch is used, then the compiler ignores all profile data not generated from code that was marked with the same session-id.

Therefore, the compiler can optimize each variant of the source's build according to how the variant is used, rather than according to an average of all uses.

Profile-Guided Optimization and the -Ov num Switch

When a .pgo file is placed on the command line, the optimization (-O) switch, by default, tries to balance between code performance and code-size considerations. It is equivalent to using the -Ov 50 switch. To optimize

solely for performance while using PGO, use the -Ov *num* switch. The `-Ov num` switch is discussed further along with optimization for space in Smaller Applications: Optimizing for Code Size.

Profile-Guided Optimization and Multiple PGO Data Sets

When using profile-guided optimization with an executable constructed from multiple source files, the use of multiple PGO data sets will result in the creation of a temporary PGO information file (`.pgi`). This file is used by the compiler and prelinker to ensure that temporary PGO files can be recreated and to identify cases where objects and PGO data sets are invalid.

The compiler reports an error if any of the PGO data files have been modified between the initial compilation of an object and any recompilation that occurs at the final link stage. To avoid this error, perform a full recompilation after running the application to generate `.pgo` data files.

When to Use Profile-Guided Optimization

PGO should be performed as the last optimization step. If the application source code is changed after gathering profile data, this profile data becomes invalid. The compiler does not use profile data when it can detect that it is inaccurate. However, it is possible to change source code in a way that is not detectable to the compiler (for example, by changing constants). You should ensure that the profile data used for optimization remains accurate.

For more details on PGO, refer to Optimization Control in the Compiler chapter.

An example application demonstrates how to use PGO in Using PGO in Function Profiling.

Using Interprocedural Optimization

To obtain the best performance, the optimizer often requires information that can only be determined by looking outside the function on which it is working. For example, it helps to know what data can be referenced by pointer parameters or whether a variable actually has a constant value. The -ipa compiler switch enables interprocedural analysis (IPA), which can make this information available. When this switch is used, the compiler is called again from the link phase to recompile the program, using additional information obtained during previous compilations.

This gathered information is stored within the object file generated during initial compilation. IPA retrieves the gathered information from the object file during linking and uses it to recompile available source files where beneficial. Because recompilation is necessary, IPA-built modules in libraries can contribute to the optimization of application sources, but do not themselves benefit from IPA, as their source is not available for recompilation.

Because it operates only at link-time, the effects of IPA are not seen if you compile with the `-S` switch. To see the assembly file when IPA is enabled, use the -save-temps switch and look at the `.s` file produced after your program has been built.

As an alternative to IPA, you can achieve many of the same benefits by adding `pragma` directives and other declarations such as `aligned()` to provide information to the compiler about how each function interacts with the rest of the program.

These directives are further described in Using the aligned() Built-in and Using Pragmas for Optimization.

The volatile Type Qualifier

The `volatile` type qualifier is used to inform the compiler that it may not make any assumptions about a variable or memory location (or a series of them), and that such variables must be read from or written to as specified and in the same order as in the source code.

Failure to use `volatile` when necessary is a common programming error that can cause an application to fail when built in Release configuration with compiler optimizations enabled. This is because the compiler assumes that all non-volatile memory is modified explicitly and does not change in a way the compiler cannot see. This assumption is used extensively during optimization, where values held in memory may not be reloaded if they do not appear to have changed. Since the cases listed below do not adhere to the compiler's assumptions, the compiler must be informed of these situations through the use of the `volatile` type qualifier.

It is essential to make the following types of objects `volatile`-qualified in your application source:

- An object that is a memory-mapped register (MMR) or a memory-mapped device.

- An object that is shared between multiple concurrent threads of execution. This includes data that is shared between processors or data written by DMA.

- An object that is modified by an asynchronous event handler (for example, a global variable modified by an interrupt handler).

- An automatic storage duration object (i.e. a local variable declared on the stack) declared in a function that calls `setjmp()` and whose value is changed between the call to `setjmp()` and a corresponding call to `longjmp()`.

Data Types

The *Scalar Data Types* table shows compiler-supported scalar data types.

Table 3-2: Scalar Data Types

Data Type	Description
Single-Word Integer Data Types: Native Arithmetic	
char	8-bit signed integer
unsigned char	8-bit unsigned integer
short	16-bit signed integer
unsigned short	16-bit unsigned integer
int	32-bit signed integer
unsigned int	32-bit unsigned integer
long	32-bit signed integer
unsigned long	32-bit unsigned integer
Fixed-Point Data Types: Native and Emulated Arithmetic	

Table 3-2: Scalar Data Types (Continued)

Data Type	Description
short fract	16-bit signed fractional
fract	16-bit signed fractional
long fract	32-bit signed fractional
unsigned short fract	16-bit unsigned fractional
unsigned fract	16-bit unsigned fractional
unsigned long fract	32-bit unsigned fractional
short accum	40-bit signed fixed-point
accum	40-bit signed fixed-point
long accum	40-bit signed fixed-point
short unsigned accum	40-bit unsigned fixed-point
unsigned accum	40-bit unsigned fixed-point
long unsigned accum	40-bit unsigned fixed-point
*Double-Word Integer Data Types:*Emulated Arithmetic	
long long	64-bit signed integer
unsigned long long	64-bit unsigned integer
Floating-Point Data Types: Emulated Arithmetic	
float	32-bit float
double	The size of the double type differs depending on the options used. If the **Double size** option or the -double-size-64 switch is used, double is a 64-bit floating-point type; otherwise, it is a 32-bit floating-point type.
long double	64-bit floating-point

The fixed-point data types fract and accum may be used by including the stdfix.h header file. Alternatively, the fractional data types fract16 and fract32 can be used, which are typedefs to integer types. Manipulation of the fract16 and fract32 data types is best done by using the built-in functions, described in Using System Support Built-In Functions.

Optimizing a struct

Memory can be saved if a struct is declared with the members ordered by size. The following example occupies 8 bytes of memory.

```
struct optimal_struct {
   char element1,element2;
   short element3;
   int element4;
};
```

However, the following example occupies 12 bytes of memory.

```
struct non_optimal_struct {
    char element1;        /* 3 bytes of padding */
    int element2;
    short element3;
    char element4;        /* 1 byte of padding  */
};
```

When the compiler generates a memory access, the access will be to a 1-, 2-, or 4-byte unit. Such accesses must be naturally aligned, meaning that 2-byte accesses must be to even addresses, and 4-byte accesses must be to addresses on a 4-byte boundary. Failure to align addresses results in a misaligned memory access exception.

The compiler is required to retain the order of members of a `struct`, and must ensure these alignment constraints are met. Therefore, by default, the compiler inserts any necessary padding to ensure that elements are aligned on their required boundaries. Padding is also inserted after the last member of a `struct` if required, to ensure that the struct's size is a multiple of the struct's strictest member alignment.

Be aware of the following additional rules of padding:

- If any member has a 4-byte alignment, the `struct` is a multiple of 4 bytes in size.

- Otherwise, if any member has a two-byte alignment, the `struct` is a multiple of two bytes in size.

- Otherwise, no end-of-struct padding is required.

Therefore, for a concrete example, if you have

```
struct non_optimal_struct test[2];
```

and if the compiler did not insert padding into the `struct non_optimal_struct`, the size of `struct non_optimal_struct` would be 8 bytes, and `test[]` array would be 16 bytes in size. Then, if

```
int x = test[1].element2;
```

this would be attempting to read an `int` (4 bytes) from a misaligned address (address of test+9), and thus a hardware exception (misaligned access) would occur.

Because the compiler adds appropriate padding in the `struct non_optimal_struct`, the `int` read will read a 4-byte aligned address (address of test+16), and the access will succeed.

As a rule of thumb, to get the smallest possible `struct`, place elements in the `struct` in the following order:

```
typedef struct efficient_struct{
    size_1_elements a,...;
    size_2_elements b,...;
    size_4_or_greater_elements c,...;
}
```

The compiler supports greater density of structs through the use of the `#pragma pack(n)` directive. This allows you to reduce the necessary padding required in structs without reordering the struct's members. There is a trade-off implied, because the compiler must still observe the architecture's address-alignment constraints. When `#pragma pack(n)` is used, it means that a `struct` member is being accessed across the required alignment boundary, and

the compiler must decompose the member into smaller, appropriately-aligned components and issue multiple accesses.

See #pragma pad (alignopt), Pragmas, in the Compiler chapter for details.

Bit-Fields

The use of bit-fields in code can reduce the amount of data storage required by an application, but will normally increase the amount of code for an application (and thus make the application slower). This is because more code is needed to access a bit-field than to access an intrinsic type (char, int, and so on). Also, bit-fields may prevent the compiler from performing optimizations that it could do on intrinsic types. However, depending on the use of bit-fields, the total data bytes plus total code bytes may be less when using bit-fields instead of intrinsic types.

The struct in the following example packs a 5-bit item, a 3-bit item, an 8-bit item, and a 16-bit item into 4 bytes.

```
struct bitf {
    int item1:5;
    int item2:3;
    char item3;
    short item4;
};
```

The array struct bitf arr[1000] would save a significant amount of data space over a non-bit-field version. However, compared to not using a bit-field, more code would be generated to access the bit-field members of the struct, and that code would be slower.

Avoiding Emulated Arithmetic

Arithmetic operations for some data types are implemented by library functions because the processor hardware does not directly support these types. Consequently, operations for these data types are far slower than native operations-sometimes by a factor of a hundred-and also produce larger code. These types are marked as "Emulated Arithmetic" in Data Types.

The hardware does not provide direct support for division, so division and modulus operations are almost always multi-cycle operations, even on integral type inputs. If the compiler has to issue a full-division operation, it usually needs to generate a call to a library function. One instance in which a library call is avoided is for integer division when the divisor is a compile-time constant and is a power of two. In this case, the compiler generates a shift instruction. Even then, a few fix-up instructions are needed after the shift if the types are signed. If you have a signed division by a power of two, consider whether you can change it to unsigned to obtain a single-instruction operation.

When the compiler has to generate a call to a library function for an arithmetic operator not supported by the hardware, performance would suffer not only because the operation takes multiple cycles, but also because the effectiveness of the compiler optimizer is reduced.

Avoid emulated arithmetic operators where possible, especially in loops, where their use can inhibit more advanced optimization techniques, such as vectorization.

Getting the Most From IPA

Interprocedural analysis (IPA) is designed to try to propagate information about the program to parts of the optimizer that can use it. This section looks at what information is useful, and how to structure your code to make this information easily accessible for analysis.

The performance features are:

- Initializing Constants Statically
- Word-Aligning Data
- Using the aligned() Built-in
- Avoiding Aliases

Initializing Constants Statically

IPA identifies variables that have only one value and replaces them with constants, resulting in a host of benefits for the optimizer's analysis. For this to happen, a variable must have a single value throughout the program. If the variable is statically initialized to zero (as are all global variables, by default) and is subsequently assigned some other value at another point in the program, then the analysis sees two values and does not consider the variable to have a constant value.

For example,

```
// BAD: IPA cannot see that val is a constant.
#include <stdio.h>
int val;          // initialized to zero

void init() {
   val = 3;       // re-assigned
}

void func() {
   printf("val %d",val);
}

int main() {
   init();
   func();
}
```

The code is better written as:

```
//GOOD: IPA knows val is 3.
#include <stdio.h>
const int val = 3;    // initialized once

void init() {
}
void func() {
```

```
   printf("val %d",val);
}
int main() {
   init();
   func();
}
```

Word-Aligning Data

To make most efficient use of the hardware, it must be continually fed with data. In many algorithms, the balance of data accesses to computations is such that, to keep the hardware fully utilized, data must be fetched with loads wider than 8 or 16 bits.

The hardware requires that references to memory be naturally aligned. Thus, 16-bit references must be at even address locations, and 32-bit references be at word-aligned addresses. Therefore, to generate the most efficient code, ensure that data buffers are word-aligned.

The compiler helps to establish the alignment of array data. Top-level arrays are allocated at word-aligned addresses, regardless of their data types. In order to do this for local arrays, the compiler also ensures that stack frames are kept word-aligned. However, arrays within structures are not aligned beyond the required alignment for their type. Consider using the #pragma align 4 directive to force the alignment of arrays in this case.

If you write programs that pass only the address of the first element of an array as a parameter, and loops that process these input arrays an element at a time, starting at element zero, then IPA should be able to establish that the alignment is suitable for full-width accesses.

Where an inner loop processes a single row of a multi-dimensional array, try to ensure that each row begins on a word boundary. In particular, two-dimensional arrays should be defined in a single block of memory rather than as an array of pointers to rows all separately allocated with malloc. It is difficult for the compiler to keep track of the alignment of the pointers in the latter case. It may also be necessary to insert dummy data at the end of each row to make the row length a multiple of four bytes.

Using the aligned() Built-in

To avoid the need to use IPA to propagate alignment, and for situations when IPA cannot guarantee the alignment (but you can), use the aligned() built-in function to assert the alignment of important pointers, meaning that the pointer points to data that is aligned.

NOTE: When adding this declaration, you are responsible for ensuring that it is valid. If the assertion is not true, the code produced by the compiler is likely to malfunction.

The assertion is particularly useful for function parameters, although you may assert that any pointer is aligned.

When compiling the following function, for example, the compiler does not know the alignment of pointers a and b if IPA is not being used.

```
// BAD: Without IPA, the compiler does not know the alignment
// of a and b.
void copy(char *a, char *b) {
   int i;
```

```
    for (i=0; i<100; i++)
        a[i] = b[i];
}
```

However, by modifying the function as follows, the compiler is told that the pointers are aligned on word boundaries.

```
// GOOD: Both pointer parameters are known to be aligned.
#include <builtins.h>

void copy(char *a, char *b) {
    int i;
    aligned(a, 4);
    aligned(b, 4);
    for (i=0; i<100; i++)
        a[i] = b[i];
}
```

To assert instead that both pointers are always aligned one `char` before a word boundary, use the following:

```
// GOOD: Both pointer parameters are known to be misaligned.
#include <builtins.h>

void copy(char *a, char *b) {
    int i;
    aligned(a+1, 4);
    aligned(b+1, 4);
    for (i=0; i<100; i++)
        a[i] = b[i];
}
```

The expression used as the first parameter to the built-in function obeys the usual C rules for pointer arithmetic. The second parameter should give the alignment in bytes as a literal constant.

Avoiding Aliases

It may seem that the iterations can be performed in any order in the following loop:

```
// BAD: a and b may alias each other.
void fn(char a[], char b[], int n) {
    int i;
    for (i = 0; i < n; ++i)
        a[i] = b[i];
}
```

But `a` and `b` are both parameters, and, although they are declared with `[]`, they are pointers that may point to the same array. When the same data may be reachable through two pointers, they are said to alias each other.

If IPA is enabled, the compiler looks at the call sites of `fn` and tries to determine whether `a` and `b` can ever point to the same array.

Even with IPA, it is easy to create what appears to the compiler as an alias. The analysis works by associating pointers with sets of variables that they may refer to at some point in the program. If the sets for two pointers intersect, then both pointers are assumed to point to the union of the two sets.

If `fn` above were called only in two places, with global arrays as arguments, then IPA would have the results shown below:

```
// GOOD: sets for a and b do not intersect:
//       a and b are not aliases.
fn(glob1, glob2, N);
fn(glob1, glob2, N);

// GOOD: sets for a and b do not intersect:
//       a and b are not aliases.
fn(glob1, glob2, N);
fn(glob3, glob4, N);

// BAD: sets intersect - both a and b may access glob1;
//       a and b may be aliases.
fn(glob1, glob2, N);
fn(glob3, glob1, N);
```

The third case arises because IPA considers the union of all calls at once, rather than considering each call individually, when determining whether there is a risk of aliasing. If each call were considered individually, IPA would have to take flow control into account and the number of permutations would significantly lengthen compilation time.

The lack of control flow analysis can also create problems when a single pointer is used in multiple contexts. For example, it is better to write

```
// GOOD: p and q do not alias.
int *p = a;
int *q = b;
   // some use of p
   // some use of q
```

than

```
// BAD: Uses of p in different contexts may alias.
int *p = a;
   // some use of p
p = b;
   // some use of p
```

because the latter may cause extra apparent aliases between the two uses.

Indexed Arrays Versus Pointers

The C language allows a program to access data from an array in two ways: either by indexing from an invariant base pointer, or by incrementing a pointer. The following two versions of vector addition illustrate the two styles.

Style 1: Using indexed arrays (indexing from a base pointer)

```
void va_ind(const short a[], const short b[], short out[], int n) {
   int i;
   for (i = 0; i < n; ++i)
      out[i] = a[i] + b[i];
}
```

Style 2: Incrementing a pointer

```
void va_ptr(const short a[], const short b[], short out[], int n) {
   int i;
   short *pout = out;
   const short *pa = a, *pb = b;
   for (i = 0; i < n; ++i)
      *pout++ = *pa++ + *pb++;
}
```

Trying Pointer and Indexed Styles

One can hope that the chosen style would not matter to the generated code, but this approach is not always the case. Sometimes, one version of an algorithm generates better optimized code than the other, but it is not always the same style that is better.

> **NOTE:** Try both pointer and indexed styles.

The pointer style introduces extra variables that compete with the surrounding code for resources during the compiler optimization analysis. Array accesses must be transformed to pointers by the compiler, which sometimes is accomplished better by hand.

The best strategy is to start with array notation. If the generated code looks unsatisfactory, try using pointers. Outside the critical loops, use the indexed style, since it is easier to understand.

Using Function Inlining

Function inlining may be used in two ways:

- By annotating functions in the source code with the `inline` keyword. In this case, function inlining is performed only when optimization is enabled.

- By turning on automatic inlining with the -Oa switch or the *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > General > Inlining* option to Automatic, automatically enabling optimization.

> **NOTE:** Inlining small frequently executed functions should improve application performance as it avoids call overheads and allows the compiler to optimize the code more effectively.

You can use the compiler's `inline` keyword to indicate that functions should have code generated inline at the point of call. Doing this avoids various costs such as program flow latencies, function entry and exit instructions, and parameter passing overheads.

Using an `inline` function also has the advantage that the compiler can optimize through the inline code and does not have to assume that scratch registers and condition states are modified by the call. Prime candidates for inlining

are small, frequently-used functions because they cause the least code-size increase while giving most performance benefit.

As an example of the usage of the `inline` keyword, the function below sums two input parameters and returns the result.

```
// GOOD: use of the inline keyword.
inline int add(int a, int b) {
   return (a+b);
}
```

Inlining has a code size-to-performance trade-off that should be considered. With `-Oa`, the compiler automatically inlines small functions where possible. If the application has a tight upper code-size limit, the resulting code-size expansion may be too great. Consider using automatic inlining in conjunction with the -Ov *num* switch or the *Optimize for code speed/size* slider to restrict inlining (and other optimizations with a code-size cost) to parts of the application that are performance-critical. It is discussed in more detail later in this chapter.

For more information, see Function Inlining in the Compiler chapter.

Using Inline asm Statements

The compiler allows use of inline `asm` statements to insert small sections of assembly into C code.

> **NOTE:** Avoid use of inline asm statements where built-in functions may be used instead.
>
> The compiler does not intensively optimize code that contains inline `asm` statements because it has little understanding about what the code in the statement does. In particular, use of an `asm` statement in a loop may inhibit useful transformations.

The compiler offers many built-in functions that generate specific hardware instructions. These are designed to allow the programmer to more finely tune the code produced by the compiler, or to allow access to system support functions. A complete list of compiler's built-in functions is given in Compiler Built-In Functions in the Compiler chapter.

Use of these built-in functions is much preferred to using inline `asm` statements. Since the compiler knows what each built-in does, it can easily optimize around them. Conversely, since the compiler does not parse `asm` statements, it does not know what they do, and so is hindered in optimizing code that uses them. Note also that errors in the text string of an `asm` statement are caught by the assembler and not by the compiler.

Examples of efficient use of built-in functions are given in Using System Support Built-In Functions.

For more information, see Inline Assembly Language Support Keyword (asm) in the Compiler chapter.

Memory Usage

The compiler, in conjunction with the use of the linker description file (`.ldf`), allows the programmer control over data placement in memory. This section describes how to best lay out data for maximum performance.

> **NOTE:** Try to put arrays into different memory sections to support efficient memory operations.

The processor hardware can support two memory operations on a single instruction line, combined with a compute instruction. Two memory operations will only complete in one cycle if the two addresses are situated in different memory blocks. If both access the same block, the processor stalls.

Consider the dot product loop below. Because data is loaded from both array a and array b in every iteration of the loop, it may be useful to ensure that these arrays are located in different blocks.

Therefore,

```
// BAD: compiler assumes that two memory accesses together  // may give a stall.
for (i=0; i<100; i++)
   sum += a[i] * b[i];
```

You could define two memory banks in the MEMORY portion of the .ldf file.

Example: MEMORY portion of the .ldf file modified to define memory banks.

```
MEMORY {
   BANK_A1 {
      TYPE(RAM) WIDTH(8)
      START(start_address_1) END(end_address_1)
   }
   BANK_A2 {
      TYPE(RAM) WIDTH(8)
      START(start_address_2) END(end_address_2)
   }
}
```

Then, you could configure the SECTIONS portion to tell the linker to place data sections in specific memory banks.

Example: SECTIONS portion of the .ldf file modified to define memory banks.

```
SECTIONS {
   bank_a1 {
      INPUT_SECTION_ALIGN(4)
      INPUT_SECTIONS($OBJECTS(bank_a1))
   } >BANK_A1
   bank_a2 {
      INPUT_SECTION_ALIGN(4)
      INPUT_SECTIONS($OBJECTS(bank_a2))
   } >BANK_A2
}
```

In the C source code, you can declare arrays with the section("section_name") pragma preceding a buffer declaration; in this case,

```
#pragma section("bank_a1")
short a[100];
#pragma section("bank_a2")
short b[100];
```

This ensures that the two array accesses in the dot product loop may occur simultaneously without incurring a stall.

The default `.ldf` files and those generated by the IDE provide a number of subdivisions within each physical memory area, so it is not usually necessary to modify your `.ldf` file directly. When possible, use the existing partitioning, so that you do not have to re-apply your changes when upgrading to a future version of the product.

Using the Bank Qualifier

The `bank` qualifier can be used to write functions that use the fact that buffers are placed in separate memory blocks.

For example, it might be useful to create a function if you would like to call `func` in different places, but always with pointers to buffers in different sections of memory.

```
// GOOD: uses bank qualifier to allow simultaneous access
// to p and q.
void func(int bank("red") *p, int bank("blue") *q) {
    // some code
}
```

The `bank` qualifier tells the compiler that the buffers are in different sections without requiring that the sections themselves be specified.

Therefore, `func` may be called with the first parameter pointing to memory in `section("bank_a1")` and the second pointing to data in `section("bank_a2")` or vice versa. You must still explicitly place the data buffers in the memory sections. The `bank` qualifier merely informs the compiler that it may assume this has been done to generate more efficient code. Refer to Memory Banks in the Compiler chapter for more information.

Improving Conditional Code

When compiling conditional statements, the compiler attempts to determine whether the condition will usually evaluate to true or to false, and will arrange for the most efficient path of execution to be that which is expected to be most commonly executed. The compiler makes these decisions based on the information in the following order:

1. If you have generated an execution profile of the function using profile-guided optimization (PGO), the compiler will compare the relative counts of the true/false paths for the branch, and will mark the path with the highest execution count as the predicted path.

2. Otherwise, if you have used one of the compiler built-in functions for explicit branch prediction (see Compiler Performance Built-In Functions in the Compiler chapter) the compiler makes the prediction as directed.

3. In the absence of all other information, the compiler will attempt to predict the branch based on heuristics and information within the source code.

This section describes:

- Using Compiler Performance Built-In Functions

- Using PGO in Function Profiling

Using Compiler Performance Built-In Functions

You can use the `expected_true` and `expected_false` built-in functions to control the compiler's optimization of conditional branches. By using these functions, you can tell the compiler which way a condition is most likely to evaluate. This influences the default flow of execution.

The following example shows two nested conditional statements.

```
if (buffer_valid(data_buffer))
   if (send_msg(data_buffer))
      system_failure();
```

If it was known that, for this example, `buffer_valid()` would usually return true, but that `send_msg()` would rarely do so, the code could be written as:

if (expected_true(buffer_valid(data_buffer))) if (expected_false(send_msg(data_buffer))) system_failure();

Example of Compiler Performance Built-in Functions

The following example project demonstrates the use of these compiler performance built-in functions:

Blackfin\Examples\No_HW_Required\ADSP-BF533\Branch_Prediction

The project loops through a section of character data, counting the different types of characters it finds. It produces three overall counts: lowercase letters, uppercase letters, and non-alphabetic characters. The effective test is as follows:

```
if (isupper(c))
   nAZ++;   // count one more uppercase letter
else if (islower(c))
   naz++;   // count one more lowercase letter
else
   nx++;    // count one more non-alphabetic character
```

The performance of the application is determined by the compiler's ability to correctly predict which of these two tests is going to evaluate as true most frequently.

In the source code for this example, the two tests are enclosed in two macros, `EXPRA(c)` and `EXPRB(c)`:

```
if (EXPRA(isupper(c)))
   nAZ++;   // count one more uppercase letter
else if (EXPRB(islower(c)))
   naz++;   // count one more lowercase letter
else
   nx++;    // count one more non-alphabetic character
```

The macros are conditionally defined according to the macro `EXPRS`, at compile-time, as shown by the ***How Macro EXPRS Affects Macros EXPRA and EXPRB*** table. By setting `EXPRS` to different values, you can see the effect the compiler performance built-in functions have on the application's overall performance. By leaving the `EXPRS` macro undefined, you can see how the compiler's default heuristics compare.

Table 3-3: How Macro EXPRS Affects Macros EXPRA and EXPRB

Value of EXPRS	EXPRA Expected to Be	EXPRB Expected to Be
Undefined	No prediction	No prediction
1	True	True
2	False	True
3	True	False
4	False	False

To use the example, do the following:

1. Import the `Branch_Prediction` project into your workspace:

    a. Select *File > Import*.

    b. Choose *General > Existing Projects into Workspace*.

    c. Ensure *Select root directory* is checked.

    d. Browse to the `Blackfin\Examples\No_HW_Required\ADSP-BF533\Branch_Prediction` directory. Click *OK*.

    e. Check the `Branch_Prediction` project.

    f. Ensure *Copy projects into workspace* is checked.

    g. Click *Finish*.

2. Build the project.

3. Create a launch configuration for the ADSP-BF533 Blackfin processor, for the executable you have just built.

4. Launch the configuration, and run the executable to completion. You will see some output on the console as the project reports the number of characters of each type found in the string. The application will also report the number of cycles used.

5. Open *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Preprocessor*.

6. In the *Defines* field, add `EXPRS=1`. Click *OK*.

7. Re-build and re-run the application. You will receive the same counts from the application, but the cycle counts will be different.

8. Try using values 2, 3, or 4 for `EXPRS` instead, and determine which combination of `expected_true()` and `expected_false()` built-in functions produces the best performance.

See Compiler Performance Built-In Functions in the Compiler chapter for more information.

Using PGO in Function Profiling

The compiler can also determine the most commonly-executed branches automatically, using profile-guided optimization (PGO). See Optimization Control in the Compiler chapter for details.

Example of Using Profile-Guided Optimization

Continuing with the same example in Using Compiler Performance Built-In Functions, PGO can determine the best settings for the branches in EXPRA(c) and EXPRB(c) (and all other parts of the source code) using profiling.

> **NOTE:** Normally, when using PGO, you would configure one or more input files as part of your data set. The application would read its inputs from these files, via the peripherals the application uses, and the data would influence the gathered profile. For this example, all the input data is embedded in the application source.

Opening the Project

To use the example, do the following:

1. Import the Branch_Prediction project into your workspace:

   a. Select *File > Import*.

   b. Choose *General > Existing Projects into Workspace*.

   c. Ensure *Select root directory* is checked.

   d. Browse to the Blackfin\Examples\No_HW_Required\ADSP-BF533\Branch_Prediction directory. Click *OK*.

   e. Check the Branch_Prediction project.

   f. Ensure *Copy projects into workspace* is checked.

   g. Click *Finish*.

2. Ensure that the *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Preprocessor > Defines* field does *not* contain a definition for EXPRS.

3. Build the project.

4. Create a launch configuration for the ADSP-BF533 Blackfin processor, for the executable you have just built.

5. Launch the configuration, and run the executable to completion. You will see some output on the console as the project reports the number of characters of each type found in the string. The application will also report the number of cycles used.

Gathering the Profile

To gather the profile on a simulator launch configuration:

1. Select *Project > Properties > C/C++ Build > Settings >Tool Settings > Compiler > Profile-guided Optimization > Prepare application to create new profile*.

2. In your launch configuration, go to the *Automatic Breakpoints* tab, and add a new software breakpoint on the label `start`.

3. Build the application, and launch it.

4. When the start breakpoint is reached, select *Target > PGO > Simulator > Start*.

5. Continue running the application, until it reaches the `__lib_prog_term label`.

6. Select *Target > PGO > Simulator > Stop and Save*.

Because the application is running on a simulator, the simulator does the work of gathering the profile, so the cycle-count will be the same as before.

To gather the profile on a hardware launch configuration:

1. Select *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Profile-guided Optimization > Gather profile using hardware*.

2. Select *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Profile-guided Optimization > Prepare application to create new profile*.

3. Build the application, and launch it.

4. Continue running the application, until it reaches the `__lib_prog_term` label.

Because the application is running on hardware, the compiler has planted additional code to gather the profile, so the cycle-count reported will be considerably higher than before. This is not a concern.

Rebuilding With the Profile

The profile will have been gathered into the file `Debug\Branch_Prediction.pgo`, within your project's directory. You now need to rebuild the application using this profile, telling the compiler to optimize the application according to execution counts for each path in the program. To do this:

1. Choose *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Profile-guided Optimization*.

   a. Ensure *Gather profile using hardware* is not selected.

   b. Ensure *Prepare application to create new profile* is not selected.

   c. Select *Optimize using existing profiles*.

   d. Add `Debug\Branch_Prediction.pgo` to the list of profiles.

2. Click the *General* (under *Compiler*) page. Ensure *Enable Optimization* is selected.

3. Rebuild the application.

Now relaunch and run your rebuilt application. You will see a lower cycle count than first reported, as the compiler has rearranged the generated code so that the most commonly-executed paths are the defaults.

Loop Guidelines

Loops are where an application ordinarily spends the majority of its time. It is therefore useful to look in detail at how to help the compiler to produce the most efficient loop code.

This section describes:

- Keeping Loops Short
- Avoiding Unrolling Loops
- Avoiding Loop-Carried Dependencies
- Avoiding Loop Rotation by Hand
- Avoiding Complex Array Indexing
- Inner Loops Versus Outer Loops
- Avoiding Conditional Code in Loops
- Avoiding Placing Function Calls in Loops
- Avoiding Non-Unit Strides
- Using 16-Bit Data Types and Vector Instructions
- Loop Control
- Using the Restrict Qualifier

Keeping Loops Short

For best code efficiency, loops should be short. Large loop bodies are usually more complex and difficult to optimize. Large loops may also require register data to be stored in memory, which decreases code density and execution performance.

Avoiding Unrolling Loops

NOTE: Do not unroll loops yourself.

Not only does loop unrolling make the program harder to read, but also prevents optimization by complicating the code for the compiler.

```
// GOOD: the compiler unrolls if it helps.
void va1(const short a[], const short b[], short c[], int n) {
    int i;
    for (i = 0; i < n; ++i) {
        c[i] = b[i] + a[i];
```

```
    }
}
// BAD: harder for the compiler to optimize.
void va2(const short a[], const short b[], short c[], int n) {
    short xa, xb, xc, ya, yb, yc;
    int i;
    for (i = 0; i < n; i+=2) {
        xb = b[i]; yb = b[i+1];
        xa = a[i]; ya = a[i+1];
        xc = xa + xb; yc = ya + yb;
        c[i] = xc; c[i+1] = yc;
    }
}
```

Avoiding Loop-Carried Dependencies

A loop-carried dependency exists when a computation in a given iteration of a loop cannot be completed without knowledge of values calculated in earlier iterations. When a loop has such dependencies, the compiler cannot overlap loop iterations. Some dependencies are caused by scalar variables that are used before they are defined in a single iteration.

However, if the loop-carried dependency is part of a *reduction* computation, the optimizer can reorder iterations. Reductions are loop computations that reduce a vector of values to a scalar value using an associative and commutative operator. A multiply and accumulate in a loop is a common example of a reduction.

```
// BAD:  loop-carried dependence in variable x.
for (i = 0; i < n; ++i)
    x = a[i] - x;
// GOOD:  loop-carried dependence is a reduction.
for (i = 0; i < n; ++i)
    x += a[i] * b[i];
```

In the first case, the scalar dependency is the subtraction operation. The variable x is modified in a manner that would give different results if the iterations were performed out of order. In contrast, in the second case, because the addition operator is associative and commutative, the compiler can perform the iterations in any order and still get the same result. Other examples of reductions are bitwise and/or and min/max operators. The existence of loop-carried dependencies that are not reductions prevents the compiler from vectorizing a loop-that is, executing more than one iteration concurrently.

Avoiding Loop Rotation by Hand

NOTE: Do not rotate loops by hand.

Programmers are often tempted to "rotate" loops in DSP code by hand, attempting to execute loads and stores from earlier or future iterations at the same time as computation from the current iteration. This technique introduces loop-carried dependencies that prevent the compiler from rearranging the code effectively. It is better to give the compiler a simpler version, and leave the rotation to the compiler.

For example,

```
// GOOD: is rotated by the compiler.
int ss(short *a, short *b, int n) {
   int sum = 0;
   int i;
   for (i = 0; i < n; i++) {
      sum += a[i] + b[i];
   }
   return sum;
}

// BAD: rotated by hand: hard for the compiler to optimize.
int ss(short *a, short *b, int n) {
   short ta, tb;
   int sum = 0;
   int i = 0;
   ta = a[i]; tb = b[i];
   for (i = 1; i < n; i++) {
      sum += ta + tb;
      ta = a[i]; tb = b[i];
   }
   sum += ta + tb;
   return sum;
}
```

Rotating the loop required adding the scalar variables `ta` and `tb` and introducing loop-carried dependencies.

Avoiding Complex Array Indexing

Other dependencies can be caused by writes to array elements. In the following loop, the optimizer cannot determine whether the load from `a` reads a value defined on a previous iteration or one that will be overwritten in a subsequent iteration.

```
// BAD: has array dependency.
for (i = 0; i < n; ++i)
   a[i] = b[i] * a[c[i]];
```

The optimizer can resolve access patterns where the addresses are expressions that vary by a fixed amount on each iteration. These are known as *induction variables*.

```
// GOOD: uses induction variables.
for (i = 0; i < n; ++i)
   a[i+4] = b[i] * a[i];
```

Inner Loops Versus Outer Loops

NOTE: Inner loops should iterate more than outer loops.

The optimizer focuses on improving the performance of inner loops because this is where most programs spend the majority of their time. It is considered a good trade-off for an optimization to slow down the code before and after a

loop to make the loop body run faster. Therefore, try to make sure that your algorithm also spends most of its time in the inner loop; otherwise it may actually run slower after optimization. If you have nested loops where the outer loop runs many times and the inner loop runs a small number of times, try to rewrite the loops so that the outer loop has fewer iterations.

Avoiding Conditional Code in Loops

If a loop contains conditional code, control-flow latencies may incur large penalties if the compiler has to generate conditional jumps within the loop. In some cases, the compiler is able to convert if-then-else and ?: constructs into conditional instructions. In other cases, it can evaluate the expression entirely outside of the loop. However, for important loops, linear code should be written where possible.

There are several techniques for removing conditional code. For example, there is hardware support for min and max. The compiler usually succeeds in transforming conditional code equivalent to min or max into the single instruction. With particularly convoluted code the transformation may be missed, in which case it is better to use min or max in the source code.

The compiler can sometimes perform the loop transformation that interchanges conditional code and loop structures. Nevertheless, instead of writing

```
// BAD: loop contains conditional code.
for (i=0; i<100; i++) {
   if (mult_by_b)
      sum1 += a[i] * b[i];
   else
      sum1 += a[i] * c[i];
}
```

it is better to write the following if this is an important loop.

```
// GOOD: two simple loops can be optimized well.
if (mult_by_b) {
   for (i=0; i<100; i++)
      sum1 += a[i] * b[i];
} else {
    for (i=0; i<100; i++)
       sum1 += a[i] * c[i];
}
```

Avoiding Placing Function Calls in Loops

The compiler usually is unable to generate a hardware loop if the loop contains a function call due to the expense of saving and restoring the context of a hardware loop. In addition, operations such as division, modulus, and some type coercions may implicitly call library functions. These are expensive operations which you should try to avoid in inner loops. For more details, see Data Types.

Avoiding Non-Unit Strides

If you write a loop, such as

```
// BAD: non-unit stride means division may be required.
for (i=0; i<n; i+=3) {
      // some code
}
```

then for the compiler to turn this into a hardware loop, it needs to work out the loop trip count. To do so, it must divide n by 3. The compiler may decide that this is worthwhile as it speeds up the loop, but division is an expensive operation. Try to avoid creating loop control variables with strides other than 1 or -1.

In addition, try to keep memory accesses in consecutive iterations of an inner loop contiguous. This is particularly applicable to multi-dimensional arrays. Therefore,

```
// GOOD: memory accesses contiguous in inner loop.
for (i=0; i<100; i++)
   for (j=0; j<100; j++)
     sum += a[i][j];
```

is likely to be better than

// BAD: loop cannot be unrolled to use wide loads. for (i=0; i<100; i++) for (j=0; j<100; j++) sum += a[j][i];

as the former is more amenable to vectorization.

Using 16-Bit Data Types and Vector Instructions

If a 16-bit, rather than 32-bit, native data type is used within a critical processing loop, the opportunities for parallel execution are increased. This is because the compiler can potentially use vector instructions, which perform simultaneous operations on multiple 16-bit values. For example, consider the simple function:

```
int func(int *a, int *b, int size) {
   int i;
   int x = 0;

   for (i= 0; i < size; i++) {
       x += a[i] + b[i];
   }
   return x;
}
```

When compiled to assembly with optimizations enabled, the compiler generates code that can potentially execute one iteration of the loop in two cycles. The equivalent function that uses the short data type is as follows:

```
short func(short *a, short *b, int size) {
   int i;
   short x = 0;

   for (i= 0; i < size; i++) {
       x += a[i] + b[i];
   }
   return x;
}
```

Here the compiler generates code that executes two iterations of the loop in two cycles with use of a vector addition. In this example, using a `short` data type doubles the performance of the loop.

Fractional arithmetic can also use vector instructions, and code generated from `fract16` built-in functions also uses these instructions as much as possible.

For more information, see Effect of Data Type Size on Code Size.

Loop Control

> **NOTE:** Use `int` types for loop control variables and array indices. Use automatic variables for loop control and loop exit test.

For loop control variables and array indices, use signed `ints` rather than other integral types. For other integral types, the C standard requires various type promotions and standard conversions that complicate the code for the compiler optimizer. Frequently, the compiler is still able to deal with such code and create hardware loops and pointer induction variables; however, it is more difficult for the compiler to optimize and may result in under-optimized code.

The same advice goes for using automatic (local) variables for loop control. It is easy for a compiler to see that an automatic scalar whose address is not taken may be held in a register during a loop. But it is not as easy when the variable is a global or a function static.

Therefore, the following code may not create a hardware loop if the compiler cannot be sure that the write into the array `a` does not change the value of the global variable. The `globvar` variable must be reloaded each time around the loop before performing the exit test.

```
// BAD: may need to reload globvar on every iteration.
for (i=0; i<globvar; i++)
   a[i] = a[i] + 1;
```

In this circumstance, the programmer can make the compiler's job easier by writing:

```
// GOOD: easily becomes a hardware loop.
int upper_bound = globvar;
for (i=0; i<upper_bound; i++)
   a[i] = a[i] + 1;
```

Using the Restrict Qualifier

The `restrict` qualifier provides one way to help the compiler resolve pointer aliasing ambiguities. Accesses from distinct restricted pointers do not interfere with each other.

The loads and stores in the following loop

```
// BAD: possible alias of arrays a and b
void copy(short *a, short *b) {
   int i;
   for (i=0; i<100; i++)
      a[i] = b[i];
}
```

may be disambiguated by writing

// GOOD: restrict qualifier tells compiler that memory // accesses do not alias

void copy(short * restrict a, short * restrict b) { int i; for (i=0; i<100; i++) a[i] = b[i]; }

Although the `restrict` keyword is particularly useful on function parameters, it can be used on any variable declaration. For example, the `copy` function may also be written as:

```
void copy(short *a, short *b) {
    int i;
    short * restrict p = a;
    short * restrict q = b;
    for (i=0; i<100; i++)
        *p++ = *q++;
}
```

Manipulating Fixed-Point and Fractional Data

Fractional data can be manipulated in different ways. This section discusses the different approaches and their advantages and limitations. In general, the styles using native fixed-point types or built-in functions are recommended, as they give you the most control over your data.

The approaches are:

- Using Integer Arithmetic to Encode Fractional Semantics

- Using Native Fixed-Point Types fract and accum

- Using Built-In Functions to Perform Fixed-Point Arithmetic

Using Integer Arithmetic to Encode Fractional Semantics

One way to manipulate fractional data involves the use of `long` promoted shifts and multiply constructs. Consider the fractional dot product algorithm. This may be written as:

```
// BAD: uses shifts to implement fractional multiplication.
long dot_product (short *a, short *b) {
    int i;
    long sum=0;
    for (i=0; i<100; i++) {
    /* this line is performance critical */
        sum += (((long)a[i]*b[i]) << 1);
    }
    return sum;
}
```

This presents problems to the optimizer. Normally, the generated code would be a multiply, followed by a shift, and then an accumulation. However, the processor hardware has a fractional multiply/accumulate instruction that performs all these tasks in one cycle.

In the example code, the compiler recognizes this idiom and replaces the multiply followed by shift with a fractional multiply. In more complicated cases, where perhaps the multiply is further separated from the shift, the compiler may not detect the possibility of using a fractional multiply.

Moreover, the transformation may in fact be invalid since it turns non-saturating integer operations into saturating fractional ones. Therefore, the results may change if the summation overflows. The transformation is enabled by default since it usually is what the programmer intended.

Using Native Fixed-Point Types fract and accum

A good way to write fixed-point arithmetic is to use the native fixed-point types `fract` and `accum`. Fixed-point arithmetic is provided on these types using the standard C operators +, −, *, and /. This means that the semantics of the arithmetic are well-defined and clear to the compiler and programmer. Moreover, there is useful run-time library support to provide further manipulations on these types. For more information, see Using Native Fixed-Point Types in the Compiler chapter.

There is an important restrictions on using these types: they are not compliant with MISRA-C, and so are not available when compiling with the −misra switch.

You could write a dot product that operates on fractional data as follows:

```
// GOOD: uses native fixed-point types to implement fractional multiplication
#include <stdfix.h>
long fract dot_product(fract *a, fract *b) {
   int i;
   accum sum=0.0k;
   for (i=0; i<100; i++) {
      /* this line is performance critical */
      sum += a[i] * b[i];
   }
   return (long fract)sum;
}
```

Using Built-In Functions to Perform Fixed-Point Arithmetic

Another way to write fractional arithmetic is to use built-in functions. This way makes the semantics of the operations clear to the compiler and encourages writing code that maps well to the Blackfin processor, since the built-in functions generally represent specific machine instructions. It also has the advantage that it may be used in MISRA-C mode, but at the expense of being less intuitive than using the native fixed-point types.

Built-in functions exist to manipulate 16- and 32-bit fractional data, as well as 40-bit values held in the accumulator registers. For more information, see Fractional Value Built-In Functions and Full-Precision Accumulator Built-In Functions in the Compiler chapter.

In the following example, a built-in function is used to multiply fractional 16-bit data.

```
// GOOD: uses built-ins to implement fractional multiplication
#include <math.h>
fract32 dot_product(fract16 *a, fract16 *b) {
   int i;
```

```
    fract32 sum=0;
    for (i=0; i<100; i++) {
        /* this line is performance critical */
        sum += mult_fr1x32(a[i],b[i]);
    }
    return sum;
}
```

Note that the `fract16` and `fract32` types used in the example above are merely `typedefs` to C integer types used by convention in standard include files. The compiler does not have any in-built knowledge of these types and treats them exactly as the integer types to which they are `typedef`'ed.

Using Built-In Functions in Code Optimization

Built-in functions, also known as *compiler intrinsics*, enable you to efficiently use low-level features of the processor hardware while programming in C. Although this section does not cover all the built-in functions available, it presents some code examples where implementation choices are available to the programmer. For more information, refer to Compiler Built-In Functions in the Compiler chapter.

Fractional Data

Built-in functions provide one way to perform arithmetic on fixed-point data. The different approaches that can be used to work with fixed-point data, including the use of built-in functions, are discussed in Manipulating Fixed-Point and Fractional Data.

Using System Support Built-In Functions

Numerous built-in functions are provided to perform low-level system management, such as system register manipulation. Built-in functions are recommended instead of inline `asm` statements.

The built-in functions cause the compiler to generate efficient inline instructions and often result in better optimization of the surrounding code at the point where they are used. Using built-in functions also results in improved code readability. For more information on supported built-in functions, refer to Compiler Built-In Functions in the Compiler chapter.

Examples of the two styles are:

```
// BAD: uses inline asm statement.
unsigned int get_cycles(void) {
    unsigned int ret_val;
    asm("%0 = CYCLES;" : "=d" (ret_val) : : );
    return ret_val;
}

// GOOD: uses sysreg.h.
#include <ccblkfn.h>
#include <sysreg.h>
unsigned int get_cycles(void) {
```

```
    return sysreg_read(reg_CYCLES);
}
```

This example reads and returns the `CYCLES` register.

Using Circular Buffers

Circular buffers are useful in DSP-style code. They can be used in several ways. Consider the C code:

```
// GOOD: the compiler knows that b is accessed
//       as a circular buffer.
for (i=0; i<1000; i++) {
   sum += a[i] * b[i%20];
}
```

The access to array `b` is a circular buffer. When optimization is enabled, the compiler produces a hardware circular buffer instruction for this access.

Consider this more complex example.

```
// BAD: may not be able to use circular buffer to access b.
for (i=0; i<1000; i+=n) {
   sum += a[i] * b[i%20];
}
```

In this case, the compiler does not know if n is positive and less than 20. If it is, the access may be correctly implemented as a hardware circular buffer. If it is greater than 20, a circular buffer increment may not yield the same results as the C code.

The programmer has two options here.

The first option is to compile with the `-force-circbuf` switch. This tells the compiler that any access of the form `a[i%n]` is to be considered as a circular buffer. Before using this switch, check that this assumption is valid for your application.

1. The value of `i` must be positive.

2. The value of `n` must be constant across the loop, and greater than zero (as the length of the buffer).

3. The value of `a` must be a constant across the loop (as the base address of the circular buffer).

4. The initial value of `i` must be such that `a[i]` refers a valid position within the circular buffer. This is because the circular buffer operations will take effect when advancing from position `a[i]` to either `a[i+m]` or `a[i-m]`, by addition or subtraction, respectively. If `a[i]` is not initially valid, access before the first advancement will not access the buffer, and `a[i+m]` and `a[i-m]` will not be guaranteed to reference the buffer after advancement.

CAUTION: Circular buffer operations (which add or subtract the buffer length to a pointer) are semantically different from `a[i%n]` (which performs a modulo operation on an index, and then adds the result to a base pointer). If you use the `-force-circbuf` switch when the above conditions are not true, the compiler generates code that does not have the intended effect.

The second (preferred) option is to use either of two built-in functions (`circindex` or `circptr`, declared in `ccblkfn.h`) to perform the circular buffering.

To inform the compiler that a circular buffer is to be used, you may write either:

```
// GOOD: explicit use of circular buffer via circindex
#include <builtins.h>
for (i=0, j=0; i<1000; i+=n) {
   sum += a[i] * b[j];
   j = circindex(j, n, 20);
}
```

or

```
// GOOD: explicit use of circular buffer via circptr
#include <builtins.h>
int *p = b;
for (i=0, j=0; i<1000; i+=n) {
   sum += a[i] * (*p);
   p = circptr(p, 4*n, b, 80);
}
```

For more information, refer to Circular Buffer Built-In Functions in the Compiler chapter.

Smaller Applications: Optimizing for Code Size

The same philosophy for producing fast code also applies to producing small code. Present the algorithm in a way that gives the optimizer clear visibility of the operations and data, hence granting it the greatest freedom to safely manipulate the code to produce small applications.

Once the program is presented in this way, the optimization strategy depends on the code size constraint that the program must obey. The first step is to optimize the application for full performance, using `-O` or `-ipa` switches. If this obeys the code size constraints, no more need be done.

The "optimize for space" switch `-Os`, which may be used in conjunction with IPA, performs every performance-enhancing transformation except those that increase code size. In addition, the `-e` linker switch (`-flags-link -e` if used from the compiler command line) may be helpful. This operation performs section elimination in the linker to remove unneeded data and code. If the code produced with the `-Os` and `-flags-link -e` switches does not meet the code size constraint, some analysis of the source code is required to try to further reduce the code size.

Note that loop transformations such as unrolling and software pipelining increase code size. But these loop transformations also give the greatest performance benefit. Therefore, in many cases compiling for minimum code size produces significantly slower code than optimizing for speed.

The compiler provides a way to balance between the two extremes of `-O` and `-Os`. This is the sliding-scale -Ov *num* switch described in the Compiler chapter. The `num` parameter may be a value between 0 and 100, where the lower value corresponds to minimum code size and the upper to maximum performance. An in-between value optimizes

frequently-executed regions of code for maximum performance, while keeping the infrequently-executed parts as small as possible.

The -Ov num switch is most reliable when using profile-guided optimization (PGO), since the execution counts of the various code regions have been measured experimentally. (See Using Profile-Guided Optimization.) Without PGO, the execution counts are estimated, based on the depth of loop nesting.

> **NOTE:** Avoid using the inline keyword to inline code for functions that are used multiple times, especially if they not very small. The -Os switch has no effect on the use of the inline keyword. It does, however, prevent automatic inlining (using the -Oa switch) from increasing the code size. Macro functions can also cause code expansion and should be used with care.

See Bit-Fields for information on how bit-fields affect code size.

Effect of Data Type Size on Code Size

For optimal performance and code size, the Blackfin architecture favors the use of 32-bit data types in control code and 16-bit data types within processing loops (see Avoiding Conditional Code in Loops), which improves the chance of vector instructions being used.

Consequently, using non-int-sized data in control code can often result in increased code size.

Short Versus Int in Control Code

```
short generate_short();
int generate_int();
void do_something();

// BAD: using short data type in control code gives
// larger code size.
void shortfunc(){
   short x;
   x=generate_short();
   x++;
   if (x==3)
      do_something();
}

// GOOD: using int data type in control code gives
// smaller code size.
void intfunc(){
   int x;
   x=generate_int();
   x++;
   if (x==3)
      do_something();
}
```

When the *Short Versus Int in Control Code* example is compiled and optimized, `shortfunc()` is slightly larger (and slower) than `intfunc()`. This is because there is no 16-bit `compare` instruction in the Blackfin architecture, and so `x` has to be sign-extended to fill a whole register before the comparison.

Using Pragmas for Optimization

Pragmas can assist optimization by allowing the programmer to make assertions or suggestions to the compiler. This section shows how they can be used to finely tune source code. Refer to Pragmas in the Compiler chapter for full details about each pragma. The emphasis of this section is to consider under what circumstances they are useful during the optimization process.

In most cases, the pragmas serve to give the compiler information that it is unable to deduce for itself. The programmer is responsible for making sure that the information given by the pragma is valid in the context in which it is used. Using a pragma to assert that a function or loop has a quality that it does not in fact have may result in incorrect code and may cause the application to malfunction.

Pragmas are advantageous because they allow code to remain portable, since pragmas are normally ignored by a compiler that does not recognize them.

The following sections describe Function Pragmas and Using Pragmas for Optimization.

Function Pragmas

Function pragmas include `#pragma alloc`, `#pragma const`, `#pragma pure`, `#pragma result_alignment`, `#pragma regs_clobbered`, and `#pragma optimize_{off|for_speed| for_space|as_cmd_line}`.

#pragma alloc

The `alloc` pragma asserts that the function behaves like the `malloc` library function. In particular, it returns a pointer to new memory that cannot alias any pre-existing buffers. In the following code, the `alloc` pragma allows the compiler to be sure that the write into buffer `p` does not modify the input buffers `q`. Therefore, the iterations of the loop may be reordered.

```
#pragma alloc
short *new_buf(void);
short *copy_buf(short *a) {
   int i;
   short * p = a;
   short * q = new_buf();
   for (i=0; i<100; i++)
      *p++ = *q++;

   return p;
}
```

#pragma const

The `const` pragma asserts to the compiler that a function does not have any side effects (such as modifying global variables or data buffers), and the result returned is only a function of the parameter values. The `const` pragma may be applied to a function prototype or definition. It helps the compiler, since two calls to the function with identical parameters always yield the same result. In this way calls to `#pragma const` functions may be hoisted out of loops if their parameters are loop independent.

#pragma pure

Like `#pragma const`, the `pure` pragma asserts to the compiler that a function does not have any side effects (such as modifying global variables or data buffers). However, the result returned may be a function of both the parameter values and any global variables. The `pure` pragma may be applied to a function prototype or definition. Two calls to the function with identical parameters yield the same result, provided that no global variables have been modified between the calls. Hence, calls to `#pragma pure` functions may be hoisted out of loops if their parameters are loop independent and no global variables are modified in the loop.

#pragma result_alignment

The `result_alignment` pragma may be used on functions that have pointer or integer results. When a function returns a pointer, the `result_alignment` pragma is used to assert that the return result always has some specified alignment. In the following example, the pragma is applied to `new_buf` to indicate that the `new_buf` function always returns buffers that are aligned on a word boundary.

```
// GOOD: uses pragma result_alignment to specify that out has
// strict alignment.
#pragma alloc
#pragma result_alignment (4)
int *new_buf(void);

int *vmul(int *a, int *b) {
   int i;
   int *out = new_buf();
   for (i=0; i<100; i++)
      out[i] = a[i] * b[i];
   return out;
}
```

For more details, see #pragma result_alignment (*n*) in the Compiler chapter. Another, more laborious way to achieve the same effect is to use `aligned()` at every call site to assert the alignment of the returned result.

#pragma regs_clobbered

The `regs_clobbered` pragma is a useful way to improve the performance of code that makes function calls. The best use of the `regs_clobbered` pragma is to increase the number of call-preserved registers available across a function call. There are two complementary ways in which this may be done.

First, suppose you have a function written in assembly that you wish to call from C source code. The `regs_clobbered` pragma may be applied to the function prototype to specify which registers are "clobbered" by the assembly function, that is, which registers may have different values before and after the function call.

The following simple assembly function adds two integers, and then masks the result to fit into 8 bits.

```
_add_mask:
   R0 = R0 + R1;
   R0 = R0.B (z);
   RTS;
._add_mask.end
```

The function does not modify the majority of the available scratch registers; thus, these may instead be used as call-preserved registers. In this way, fewer spills to the stack are needed in the caller function.

Using the following prototype, the compiler is told which registers are modified by a call to the `add_mask` function. Registers not specified by the pragma are assumed to preserve their values across such a call, and the compiler may use these spare registers to its advantage when optimizing the call sites.

```
// GOOD: uses regs_clobbered to increase call-preserved  // register set.
#pragma regs_clobbered "R0, ASTAT"
int add_mask(int, int);
```

The pragma is also powerful when all of the source code is written in C. In the above example, a C implementation can be:

```
// BAD: function thought to clobber entire volatile register set.
int add_mask(int a, int b) {
   return ((a+b)&255);
}
```

Since this function does not need many registers when compiled, it can be defined using the following code to ensure that any other registers aside from `R0` and the condition codes are not modified by the function.

```
// GOOD: function compiled to preserve most registers.
#pragma regs_clobbered "R0, CCset"
int add_mask(int a, int b) {
   return ((a+b)&255);
}
```

If other registers are used in the compilation of the function, they are saved and restored during the function prologue and epilogue.

In general, it is not helpful to specify any of the condition codes as call-preserved, as they are difficult to save and restore and are usually clobbered by any function. Moreover, it is usually of limited benefit to keep them live across a function call. Therefore, it is better to use `CCset` (all condition codes) rather than `ASTAT` in the clobbered set above.

For more information, refer to #pragma regs_clobbered *string* *string* in the Compiler chapter.

#pragma optimize_{off|for_speed|for_space|as_cmd_line}

The `optimize` pragmas may be used to change the optimization setting on a function-by-function basis. In particular, it may be useful to optimize functions that are rarely called (for example, error handling code) for space (`#pragma optimize_for_space`), whereas functions critical to performance should be compiled for maximum speed (using `#pragma optimize_for_speed`). The `#pragma optimize_off` is useful for debugging specific functions without increasing the size or decreasing the performance of the overall application unnecessarily.

`#pragma optimize_as_cmd_line` resets the optimization settings to those specified on the `ccblkfn` command line when the compiler was invoked. Refer to General Optimization Pragmas in the Compiler chapter for more information.

Loop Optimization Pragmas

Many pragmas are targeted towards helping to produce optimal code for inner loops. These are the `loop_count`, `no_vectorization`, `vector_for`, `all_aligned`, `different_banks`, and `no_alias` pragmas.

#pragma loop_count

The `loop_count` pragma enables the programmer to inform the compiler about a loop's iteration count. The compiler is able to make more reliable decisions about the optimization strategy for a loop when it knows the iteration count range. If you know that the loop count is always a multiple of a constant, this can also be useful, as it allows a loop to be partially unrolled or vectorized without the need for conditionally-executed iterations. Knowledge of the minimum trip count may allow the compiler to omit the guards that are usually required after software pipelining. (A *guard* is code generated by the compiler to test a condition at runtime rather than at compile-time.) Any of the unknown parameters of the pragma may be left blank.

The following is an example of the `loop_count` pragma:

```
// GOOD: the loop_count pragma gives the compiler helpful
// information to assist optimization.
#pragma loop_count(/*minimum*/ 40, /*maximum*/ 100, /*modulo*/ 4)
for (i=0; i<n; i++)
   a[i] = b[i];
```

For more information, refer to #pragma loop_count(*min, max, modulo*) , in the Compiler chapter.

#pragma no_vectorization

Vectorization (executing more than one iteration of a loop in parallel) can slow down loops with small iteration counts, since a loop prologue and epilogue are required. The `no_vectorization` pragma can be used directly above a `for` or `do` loop to instruct the compiler not to vectorize the loop, or directly before a function to disable vectorization for all loops in the function.

#pragma vector_for

The `vector_for` pragma is used to help the compiler resolve dependencies that prevent it from vectorizing a loop. It tells the compiler that all iterations of the loop may be run in parallel with each other, subject to

rearrangement of reduction expressions in the loop. In other words, there are no loop-carried dependencies except reductions. An optional parameter, n, may be given in parentheses to indicate that only n iterations of the loop may be run in parallel. The parameter must be a literal value.

For example, the following cannot be vectorized if the compiler cannot tell that array b does not alias array a.

```
// BAD: cannot be vectorized due to possible alias between  // a and b.
for (i=0; i<100; i++)
   a[i] = b[i] + a[i-4];
```

But the vector_for pragma may be added to tell the compiler that in this case four iterations may be executed concurrently, as follows:

```
// GOOD: pragma vector_for disambiguates alias.
#pragma vector_for (4)
for (i=0; i<100; i++)
   a[i] = b[i] + a[i-4];
```

Note that this pragma does not force the compiler to vectorize the loop. The optimizer checks various properties of the loop and does not vectorize it if it believes that it is unsafe or cannot deduce information necessary to carry out the vectorization transformation. The pragma assures the compiler that there are no loop-carried dependencies, but other properties of the loop may prevent vectorization.

In cases where vectorization is impossible, the information given in the assertion made by vector_for may still aid other optimizations.

For more information, refer to #pragma vector_for in the Compiler chapter.

#pragma all_aligned

The all_alignedpragma is used as shorthand for multiple aligned() assertions. Prefixing a for loop with this pragma asserts that every pointer variable in the loop is aligned on a word boundary at the beginning of the first iteration. Thus, adding the pragma to the following loop

```
// GOOD: uses all_aligned to inform compiler of alignment of  // a and b.
#pragma all_aligned
for (i=0; i<100; i++)
   a[i] = b[i];
```

is equivalent to writing

```
// GOOD: uses aligned() to give alignment of a and b.
#include <builtins.h>
aligned(a, 4);
aligned(b, 4);
for (i=0; i<100; i++)
   a[i] = b[i];
```

In addition, the all_aligned pragma may take an optional literal integer argument, n, in parentheses. This tells the compiler that all pointer variables are aligned on a word boundary at the beginning of the nth iteration. Note that the iteration count begins at zero.

Therefore,

```
// GOOD: uses all_aligned to inform compiler of alignment  // of a and b.
#pragma all_aligned (3)
for (i=99; i>=0; i--)
   a[i] = b[i];
```

is equivalent to

```
// GOOD: uses aligned() to give alignment of a and b.
#include <builtins.h>
aligned(a+96, 4);
aligned(b+96, 4);
for (i=99; i>=0; i--)
   a[i] = b[i];
```

For more information, refer to #pragma all_aligned in Using the aligned() Built-in and the Compiler chapter.

#pragma different_banks

The different_banks pragma is used as shorthand for declaring multiple pointer types with different bank qualifiers. It asserts that any two independent memory accesses in the loop may be issued together without incurring a stall.

Therefore, writing the following allows a single instruction loop to be created if it is known that a and b do not alias each other.

```
// GOOD: uses different banks to allow simultaneous accesses
//        to a and b.
#pragma different_banks
for (i=0; i<100; i++)
   a[i] = b[i];
```

See #pragma different_banks in the Compiler chapter for more information.

#pragma no_alias

When immediately preceding a loop, the no_alias pragma asserts that no load or store in the loop accesses the same memory. This helps to produce shorter loop kernels because it permits instructions in the loop to be rearranged more freely. See #pragma no_alias in the Compiler chapter for more information.

Optimization Switches

The *C/C++ Compiler Optimization Switches* table lists compiler switches useful during the optimization process. Refer to the Compiler chapter for details.

Table 3-4: C/C++ Compiler Optimization Switches

Switch	Description
-const-read-write	Specifies that data accessed via a pointer to const data may be modified elsewhere

Table 3-4: C/C++ Compiler Optimization Switches (Continued)

Switch	Description
`-flags-link -e`	Specifies linker section elimination
`-force-circbuf`	Treats array references of the form `array[i%n]` as circular buffer operations
`-ipa`	Turns on inter-procedural optimization. Implies use of `-O`. May be used in conjunction with `-Os` or `-Ov`.
`-no-fp-associative`	Does not treat floating-point multiply and addition as an associative
`-O`	Enables code optimizations and optimizes the file for speed
`-Os`	Optimizes the file for size
`-Ov num`	Controls speed vs. size optimizations (sliding scale)
`-save-temps`	Saves intermediate files (for example, `.s`)

How Loop Optimization Works

Loop optimization is important to overall application performance, because any performance gain achieved within the body of a loop reaps a benefit for every iteration of that loop. This section provides an introduction to some of the concepts used in loop optimization, helping you to use the compiler features in this chapter.

This section contains:

- Terminology

- Loop Optimization Concepts

- Optimizer Example

Terminology

This section describes terms that have particular meanings for compiler behavior.

Clobbered

A register is *clobbered* if its value is changed so that the compiler cannot usefully make assumptions about register's new contents.

For example, when the compiler generates a call to an external function, the compiler considers all caller-preserved registers to be clobbered by the called function. Once the called function returns, the compiler cannot make any assumptions about the values of those registers. This is why they are called *caller-preserved*. If the caller needs the values in those registers, the caller must preserve them itself.

The set of registers clobbered by a function can be changed using `#pragma regs_clobbered`, and the set of registers changed by a gnu `asm` statement is determined by the clobber part of the `asm` statement.

Live

A register is *live* if it contains a value needed by the compiler, and thus cannot be overwritten by a new assignment to that register. For example, to do "A = B + C", the compiler might produce:

```
reg1 = load B         // reg1 becomes live
reg2 = load C         // reg2 becomes live
reg1 = reg1 + reg2    // reg2 ceases to be live;
                      // reg1 still live, but with a different value
store reg1 to A       // reg1 ceases to be live
```

Liveness determines which registers the compiler may use. In this example, since `reg1` is used to load `B`, and that register must maintain its value until the addition, `reg1` cannot also be used to load the value of `C`, unless the value in `reg1` is first stored elsewhere.

Spill

When a compiler needs to store a value in a register, and all usable registers are already live, the compiler must store the value of one of the registers to temporary storage (the stack). This *spilling* process prevents the loss of a necessary value.

Scheduling

Scheduling is the process of re-ordering the program instructions to increase the efficiency of the generated code but without changing the program's behavior. The compiler attempts to produce the most efficient schedule.

Loop Kernel

The *loop kernel* is the body of code that is executed once per iteration of the loop. It excludes any code required to set up the loop or to finalize it after completion.

Loop Prolog

A *loop prolog* is a sequence of code required to set the machine into a state whereby the loop kernel can execute. For example, the prolog may pre-load some values into registers ready for use in the loop kernel. Not all loops need a prolog.

Loop Epilog

A *loop epilog* is a sequence of code responsible for finalizing the execution of a loop. After each iteration of the loop kernel, the machine will be in a state where the next iteration can begin efficiently. The epilog moves values from the final iteration to where they need to be for the rest of the function to execute. For example, the epilog might save values to memory. Not all loops need an epilog.

Loop Invariant

A *loop invariant* is an expression that has the same value for all iterations of a loop. For example,

```
int i, n = 10;
for (i = 0; i < n; i++) {
   val += i;
}
```

The variable n is a loop invariant. Its value is not changed during the body of the loop, so n will have the value 10 for every iteration of the loop.

Hoisting

When the optimizer determines that some part of a loop is computing a value that is actually a loop invariant, it may move that computation to before the loop. This *hoisting* prevents the same value from being re-computed for every iteration.

Sinking

When the optimizer determines that some part of a loop is computing a value that is not used until the loop terminates, the compiler may move that computation to after the loop. This *sinking process* ensures the value is only computed using the values from the final iteration.

Loop Optimization Concepts

The compiler optimizer focuses considerable attention on program loops, as any gain in the loop's performance reaps the benefits on every iteration of the loop. The applied transformations can produce code that appears to be substantially different from the structure of the original source code. This section provides an introduction to the compiler's loop optimization, to help you understand why the code might be different.

The following examples are presented in terms of a hypothetical machine. This machine is capable of issuing up to two instructions in parallel, provided one instruction is an arithmetic instruction, and the other is a load or a store. Two arithmetic instructions may not be issued at once, nor may two memory accesses:

```
t0 = t0 + t1;              // valid: single arithmetic
t2 = [p0];                 // valid: single memory access
[p1] = t2;                 // valid: single memory access
t2 = t1 + 4, t1 = [p0];    // valid: arithmetic and memory
t5 += 1, t6 -= 1;          // invalid: two arithmetic
[p3] = t2, t4 = [p5];      // invalid: two memory
```

The machine can use the old value of a register and assign a new value to it in the same cycle, For example,

```
t2 = t1 + 4, t1 = [p0];    // valid: arithmetic and memory
```

The value of t1 on entry to the instruction is the value used in the addition. On completion of the instruction, t1 contains the value loaded via the p0 register.

The examples will show START LOOP N and END LOOP, to indicate the boundaries of a loop that iterates N times. (The mechanisms of the loop entry and exit are not relevant.)

Software Pipelining

Software pipelining is analogous to hardware pipelining used in some processors. Whereas hardware pipelining allows a processor to start processing one instruction before the preceding instruction has completed, software pipelining allows the generated code to begin processing the next iteration of the original source-code loop before the preceding iteration is complete.

Software pipelining makes use of a processor's ability to multi-issue instructions. Regarding known delays between instructions, it also schedules instructions from later iterations where there is spare capacity.

Loop Rotation

Loop rotation is a common technique of achieving software pipelining. It changes the logical start and end positions of the loop within the overall instruction sequence, to allow a better schedule within the loop itself. For example, this loop:

```
START LOOP N
    A
    B
    C
    D
    E
END LOOP
```

could be rotated to produce the following loop:

```
A
B
C
START LOOP N-1
    D
    E
    A
    B
    C
END LOOP
D
E
```

The order of instructions in the loop kernel is now different. It still circles from instruction E back to instruction A, but now it starts at D, rather than A. The loop also has a prolog and epilog added, to preserve the intended order of instructions. Since the combined prolog and epilog make up a complete iteration of the loop, the kernel is now executing N-1 iterations, instead of N.

Another example-consider the following loop:

```
START LOOP N
    t0 += 1
    [p0++] = t0
END LOOP
```

This loop has a two-cycle kernel. While the machine could execute the two instructions in a single cycle-an arithmetic instruction and a memory access instruction-to do so would be invalid, because the second instruction depends upon the value computed in the first instruction. However, if the loop is rotated, we get:

```
t0 += 1
START LOOP N-1
    [p0++] = t0
    t0 += 1
```

```
END LOOP
[p0++] = t0
```

The value being stored is computed in the previous iteration (or before the loop starts, in the prolog). This allows the two instructions to be executed in a single cycle:

```
t0 += 1
START LOOP N-1
  [p0++] = t0, t0 += 1
END LOOP
[p0++] = t0
```

Rotating the loop has presented an opportunity by which the kth iteration of the original loop is starting (t0 += 1) while the (k-1)th iteration is completing ([p0++] = t0). As a result, rotation has achieved software pipelining, and the performance of the loop is doubled.

Notice that this process has changed the structure of the program slightly. Suppose that the loop construct always executes the loop at least once; that is, it is a 1..N count. Then if N==1, changing the loop to be N-1 would be problematic. In this example, the compiler inserts a conditional jump around the loop construct for the circumstances where the compiler cannot guarantee that N > 1:

```
t0 += 1
IF N == 1 JUMP L1;
    START LOOP N-1
        [p0++] = t0, t0 += 1
    END LOOP
L1:
    [p0++] = t0
```

Loop Vectorization

Loop vectorization is another transformation that allows the generated code to execute more than one iteration in parallel. However, vectorization is different from software pipelining. Where software pipelining uses a different ordering of instructions to get better performance, vectorization uses a different set of instructions. These vector instructions act on multiple data elements concurrently to replace multiple executions of each original instruction.

For example, consider the following dot product loop:

```
int i, sum = 0;
for (i = 0; i < n; i++) {
    sum += x[i] * y[i];
}
```

This loop walks two arrays, reading consecutive values from each, multiplying them and adding the result to the ongoing sum. This loop has these important characteristics:

- Successive iterations of the loop read from adjacent locations in the arrays.

- The dependency between successive iterations is the summation, a commutative operation.

- Operations such as load, multiply and add are often available in parallel versions on embedded processors.

These characteristics allow the optimizer to vectorize the loop so that two elements are read from each array per load, two multiplies are done, and two totals maintained. The vectorized loop would be:

```
t0 = t1 = 0
START LOOP N/2
t2 = [p0++] (Wide)     // load x[i] and x[i+1]
t3 = [p1++] (Wide)     // load y[i] and y[i+1]
t0 += t2 * t3 (Low), t1 += t2 * t3 (High)   // vector mulacc
END LOOP
t0 = t0 + t1           // combine totals for low and high
```

Vectorization is most efficient when all the operations in the loop can be expressed in terms of parallel operations. Loops with conditional constructs in them are rarely vectorizable, because the compiler cannot guarantee that the condition will evaluate in the same way for all the iterations being executed in parallel.

Vectorization is also affected by data alignment constraints and data access patterns. Data alignment affects vectorization because processors often constrain loads and stores to be aligned on certain boundaries. While the unvectorized version will guarantee this, the vectorized version imposes a greater constraint that may not be guaranteed. Data access patterns affect vectorization because memory accesses must be contiguous. If a loop accessed every tenth element, for example, then the compiler would not be able to combine the two loads for successive iterations into a single access.

Vectorization divides the generated iteration count by the number of iterations being processed in parallel. If the trip count of the original loop is unknown, the compiler will have to conditionally execute some iterations of the loop.

If the compiler cannot determine whether the loop is "vectorizable" at compile-time and the speed/space optimization settings allow it, the compiler will generate vectorized and non-vectorized versions of the loop. It will select between the two at run-time. This allows for considerable performance improvements, at the expense of code-size and an initial set-up cost.

> **NOTE:** Vectorization and software pipelining are not mutually exclusive: the compiler may vectorize a loop and then use software pipelining to obtain better performance.

Modulo Scheduling

Loop rotation, as described earlier, is a simple software-pipelining method that can often improve loop performance, but more complex examples require a more advanced approach. The compiler uses a popular technique known as *modulo scheduling* which can produce more efficient schedules for loops than simple loop rotation. (See also Modulo Scheduling Information.)

Modulo scheduling is used to schedule innermost loops without control flow. A modulo-scheduled loop is described using the following parameters:

- Initiation interval (II): the number of cycles between initiating two successive iterations of the original loop.

- Minimum initiation interval due to resources (res MII): a lower limit for the initiation interval (II); an II lower than this would mean at least one of the resources being used at greater capacity than the machine allows.

- Minimum initiation interval due to recurrences (rec MII): an instruction cannot be executed until earlier instructions on which it depends have also been executed. These earlier instructions may belong to a previous loop iteration. A cycle of such dependencies (a recurrence) imposes a minimum number of cycles for the loop.

- Stage count (SC): the number of initiation intervals until the first iteration of the loop has completed. This is also the number of iterations in progress at any time within the kernel.

- Modulo variable expansion unroll factor (MVE unroll): the number of times the loop has to be unrolled to generate the schedule without overlapping register lifetimes.

- Trip count: the number of times the loop kernel iterates.

- Trip modulo: a number that is known to divide the trip count.

- Trip maximum: an upper limit for the trip count.

- Trip minimum: a lower limit for the trip count.

Understanding these parameters will allow you to interpret the generated code more easily. The compiler's assembly annotations use these terms, so you can examine the source code and the generated instructions, to see how the scheduling relates to the original source. See Assembly Optimizer Annotations for more information.

Modulo scheduling performs software pipelining by:

- Ordering the original instructions in a sequence (for simplicity referred to as the *base schedule*) that can be repeated after an interval known as the *initiation interval* ("II");

- Issuing parts of the base schedule belonging to successive iterations of the original loop, in parallel.

For the purposes of this discussion, all instructions will be assumed to require only a single cycle to execute; on a real processor, stalls affect the initiation interval, so a loop that executes in II cycles may have fewer than II instructions.

Initiation Interval (II) and the Kernel

Consider the loop:

```
START LOOP N
  A
  B
  C
  D
  E
  F
  G
  H
END LOOP
```

Now consider that the compiler finds a new order for A, B, C, D, E, F, G, H grouping; some of them on the same cycle so that a new instance of the sequence can be started every two cycles. Say this base schedule is given in the *Base Schedule* table, where I1, I2, ..., I8 are A, B, ..., H reordered. Albeit a valid schedule for the original loop, the base schedule is not the final modulo schedule; it may not even be the shortest schedule of the original

loop. However the base schedule is used to obtain the modulo schedule, by being able to initiate it every II=2 cycles, as seen in the ***Obtaining the Modulo Schedule*** table.

Table 3-5: Base Schedule

Cycle	Instructions
1	I1
2	I2, I3
3	I4, I5
4	I6
5	I7
6	I8

Table 3-6: Obtaining the Modulo Schedule (by repeating the base schedule every II=2 cycles, assuming a maximum of 4 instructions executed in parallel per cycle)

Cycle	Iteration 1	Iteration 2	Iteration 3	Iteration 4
1	I1			
2	I2, I3			
3	I4, I5	I1		
4	I6	I2, I3		
5	*I7*	*I4, I5*	*I1*	
6	*I8*	*I6*	*I2, I3*	
7		I7	I4, I5	I1
8		I8	I6	I2, I3
9			I7	I4, I5
10			I8	I6

Starting at cycle 5, the pattern in the ***Loop Kernel*** table repeats every 2 cycles. This repeating pattern, the kernel, represents the modulo-scheduled loop.

Table 3-7: Loop Kernel (N>=3)

Cycle	Iteration N-2 (Last Stage)	Iteration N-1 (2nd Stage)	Iteration N (1st Stage)
II*N-1	I7	I4, I5	I1
II*N	I8	I6	I2, I3

The initiation interval has the value II=2, because iteration `i+1` can start two cycles after the cycle on which iteration `i` starts. This way, one iteration of the original loop is initiated every II cycles, running in parallel with previous, unfinished iterations.

The initiation interval of the loop indicates several important characteristics of the schedule for the loop:

- The loop kernel will be II cycles in length.

- A new iteration of the original loop will start every II cycles. An iteration of the original loop will end every II cycles.

- The same instruction will execute on cycle `c` and on cycle `c+II` (hence the name modulo schedule).

Finding a modulo schedule implies finding a base schedule and an II such that the base schedule can be initiated every II cycles.

If the compiler can reduce the value for II, it can start the next iteration sooner, and thus increase the performance of the loop: The lower the II, the more efficient the schedule. However, the II is limited by a number of factors, including:

- The machine resources required by the instructions in the loop.

- The data dependencies and stalls between instructions.

These limiting factors are examined in:

- Minimum Initiation Interval Due to Resources (Res MII)

- Minimum Initiation Interval Due to Recurrences (Rec MII)

- Stage Count (SC)

- Variable Expansion and MVE Unroll

- Trip Count

Minimum Initiation Interval Due to Resources (Res MII)

The first factor that limits II is machine resource usage. Let's start with the simple observation that the kernel of a modulo-scheduled loop contains the same set of instructions as the original loop.

Assume a machine that can execute up to four instructions in parallel. If the loop has 8 instructions, then it requires a minimum of two lines in the kernel, since there can be at most 4 instructions on a line. This implies II has to be at least 2, and we can tell this without having found a base schedule for the loop, or even knowing what the specific instructions are.

Consider another example where the original loop contains 3 memory accesses to be scheduled on a machine that supports at most 2 memory accesses per cycle. This implies at least 2 cycles in the kernel, regardless of the rest of the instructions.

Given a set of instructions in a loop, we can determine a lower bound for the II of any modulo schedule for that loop based on resources required. This lower bound is called the *Resource-based Minimum Initiation Interval* (Res MII).

Minimum Initiation Interval Due to Recurrences (Rec MII)

A less obvious limitation for finding a low II are cycles in the data dependencies between instructions.

Assume that the loop to be scheduled contains (among others) the instructions:

```
i3: t3=t1+t5; // t5 carried from the previous iteration i5:
t5=t1+t3;
```

Assume each line of instructions takes 1 cycle. If `i3` is executed at cycle `c`, then `t3` is available at cycle `c+1` and `t5` cannot be computed earlier than `c+1` (because it depends on `t3`), and similarly the next time we compute `t3` cannot be earlier than `c+2`. Thus, if we execute `i3` at cycle `c`, the next time we can execute `i3` again cannot be earlier than `c+2`. But for any modulo schedule, if an instruction is executed at cycle `c`, the next iteration will execute the same instruction at cycle `c+II`. Therefore, II has to be at least 2 due to the circular data dependency path `t3->t5->t3`.

This lower bound for II, given by circular data dependencies (recurrences) is called the "Minimum Initiation Interval Due to Recurrences" (Rec MII), and the data dependency path is called "loop carry path". There can be any number of loop carry paths in a loop, including none, and they are not necessarily disjoint.

Stage Count (SC)

The kernel in the *Loop Kernel* table (see Initiation Interval (II) and the Kernel) is formed of instructions which belong to three distinct iterations of the original loop: {`I7`,`I8`} end the "oldest" iteration-in other words they belong to the iteration started the longest time before the current cycle; {`I4`,`I5`,`I6`} belong to the next oldest initiated iteration, and so on. {`I1`,`I2`,`I3`} are the beginning of the youngest iteration.

The number of iterations of the original loop in progress at any time within the kernel is called the "Stage Count" (SC). This is also the number of initiation intervals until the first iteration of the loop completes. In our example, SC=3.

The final schedule requires peeling a few instructions (the prolog) from the beginning of the first iteration and a few instructions (the epilog) from the end of the last iteration in order to preserve the structure of the kernel. This reduces the trip count from N to N-(SC-1):

```
I1;                         // prolog
I2,I3;                      // prolog
I4,I5,   I1;                // prolog
I6,      I2,I3;             // prolog
LOOP N-2                    // i.e. N-(SC-1), where SC=3
  I7,       I4,I5,   I1;    // kernel
  I8,       I6,      I2,I3; // kernel
END LOOP
          I7,       I4, I5; // epilog
          I8,       I6;     // epilog
                    I7;     // epilog
                    I8;     // epilog
```

Another way of viewing the modulo schedule is to group instructions into stages as in the *Instructions Grouped into Stages* table, where each stage is viewed as a vector of height II=2 of instruction lists (that represent parts of instruction lines).

Table 3-8: Instructions Grouped into Stages

Stage Count	Instructions
SC0	I1, I2, I3
SC1	I4, I5, I6
SC2	I7, I8

Now the schedule can be viewed as:

```
SC0                // prolog
SC1    SC0         // prolog
LOOP (N-2)         // That is N-(SC-1), where SC=3
  SC2    SC1    SC0 // kernel
END LOOP
      SC2    SC1    // epilog
             SC2    // epilog
```

where, for example, SC2 SC1 is the 2-line vector obtained from concatenating the lists in SC2 and SC1.

Variable Expansion and MVE Unroll

There is one more issue to address for modulo schedule correctness.

Consider the sequence of instructions in the *Problematic Instance* table. The *Base Schedule Applied* table shows the base schedule that is an instance of the one in the *Base Schedule* table, and the *Modulo Schedule* table shows the corresponding modulo schedule with II=2.

Table 3-9: Problematic Instance

Generic Instruction	Specific Instance
I1	t1=[p1++]
I2	t2=[p2++]
I3	t3=t1+t5
I4	t4=t2+1
I5	t5=t1+t3
I6	t6=t4*t5
I7	t7=t6*t3
I8	[p8++]=t7

Table 3-10: Base Schedule Applied (from the Base Schedule table to instances in the Problematic Instance table)

1	t1=[p1++]
2	t2=[p2++],t3=t1+t5
3	t4=t2+1,t5=t1+t3

Table 3-10: Base Schedule Applied (from the Base Schedule table to instances in the Problematic Instance table) (Continued)

4	t6=t4*t5
5	t7=t6*t3
6	[p8++]=t7

Table 3-11: Modulo Schedule (broken by overlapping lifetimes of t3)

	Iteration 1	*Iteration 2*	*Iteration 3 ...*
1	t1=[p1++]		
2	t2=[p2++],t3=t1+t5		
3	t4=t2+1,t5=t1+t3	t1=[p1++]	
4	t6=t4*t5	t2=[p2++],t3=t1+t5	
5	*t7=t6*t3*	*t4=t2+1,t5=t1+t3*	*t1=[p1++]*
6	*[p8++]=t7*	*t6=t4*t5*	*t2=[p2++],t3=t1+t5*
7		t7=t6*t3	t4=t2+1,t5=t1+t3
8		[p8++]=t7	t6=t4*t5
9			t7=t6*t3
10			[p8++]=t7

There is a problem with the schedule in the ***Modulo Schedule*** table: t3 defined in the fourth cycle (second column in the table) is used on the fifth cycle (first column); however, the intended use was of the value defined on the second cycle (first column). In general, the value of t3 used by t7=t6*t3 in the kernel will be the one defined in the previous cycle, instead of the one defined 3 cycles earlier, as intended. Thus, if the compiler were to use this schedule as-is, it would be clobbering the live value in t3.

The lifetime of each value loaded into t3 is 3 cycles, but the loop's initiation interval is only 2, so the lifetimes of t3 from different iterations overlap.

The compiler fixes this by duplicating the kernel as many times as needed to exceed the longest lifetime in the base schedule, then renaming the variables that clash-in this case, just t3.

In the ***Modulo Schedule Corrected*** table, we see that the length of the new loop body is 4, greater than the lifetimes of the values in the loop.

So the loop becomes:

```
t1=[p1++];
t2=[p2++],t3=t1+t5;
t4=t2+1,t5=t1+t3,    t1=[p1++];
t6=t4*t5,            t2=[p2++],t3_2=t1+t5;
LOOP (N-2)/2
t7=t6*t3,            t4=t2+1,t5=t1+t3_2,    t1=[p1++];
[p8++]=t7,           t6=t4*t5,             t2=[p2++],t3=t1+t5;
```

```
                        t7=t6*t3_2,              t4=t2+1,t5=t1+t3,t1=[p1++];
                        [p8++]=t7,               t6=t4*t5, t2=[p2++],t3_2=t1+t5;
END LOOP
                        t7=t6*t3,                t4=t2+1,t5=t1+t3_2;
                        [p8++]=t7,               t6=t4*t5;
                                                 t7=t6*t3_2;
                                                 [p8++]=t7;
```

Table 3-12: Modulo Schedule Corrected (by variable expansion: t3 and t3_2)

	Iteration 1	Iteration 2	Iteration 3	Iteration 4 ...
1	t1=[p1++]			
2	t2=[p2++],t3=t1+t5			
3	t4=t2+1,t5=t1+t3	t1=[p1++]		
4	t6=t4*t5	t2=[p2++],t3_2=t1+t5		
5	*t7=t6*t3*	*t4=t2+1,t5=t1+t3_2*	*t1=[p1++]*	
6	*[p8++]=t7*	*t6=t4*t5*	*t2=[p2++],t3=t1+t5*	
7		*t7=t6*t3_2*	*t4=t2+1,t5=t1+t3*	*t1=[p1++]*
8		*[p8++]=t7*	*t6=t4*t5*	*t2=[p2++],t3_2=t1+t5*
9			t7=t6*t3	t4=t2+1,t5=t1+t3_2
10			[p8++]=t7	t6=t4*t5
11				t7=t6*t3_2
12				[p8++]=t7

This process of duplicating the kernel and renaming colliding variables is called variable expansion, and the number of times the compiler duplicates the kernel is referred to as the modulo variable expansion factor (MVE). Conceptually we use different set of names, *register sets*, for successive iterations of the original loop in progress in the unrolled kernel (in practice we rename just the conflicting variables, see the following >*Instructions After Modulo Variable Expansion* table). In terms of reading the code, this means that a single iteration of the loop generated by the compiler will be processing more than one iteration of the original loop. Also, the compiler will be using more registers to allow the iterations of the original loop to overlap without clobbering the live values.

In terms of stages:

```
SC0                                   // prolog
SC1    SC0_2                          // prolog
LOOP (N-2)/2                          // That is N-(SC-1)/MVE, where SC=3, MVE=2
SC2    SC1_2    SC0                   // kernel
       SC2_2    SC1    SC0_2          // kernel
END LOOP
                SC2    SC1_2          // epilog
                SC2_2                 // epilog
```

where SCN_2 is SCN subject to renaming; in our case, only occurrences of t3 are renamed as t3_2 in SCN_2.

In terms of instructions:

```
I1;                                // prolog
I2,I3;                             // prolog
I4,I5,    I1_2;                    // prolog
I6,       I2_2,I3_2;              // prolog
LOOP(N-2)/2                        // That is N-(SC-1)/MVE, where SC=3, MVE=2
I7,   I4_2,I5_2,        I1;       // kernel
I8,   I6_2,    I2,I3;             // kernel
      I7_2,    I4,I5,   I1_2;     // kernel
      I8_2,    I6,      I2_2,I3_2; // kernel
END LOOP
            I7,   I4_2,I5_2;      // epilog
            I8,   I6_2;          // epilog
            I7_2;                 // epilog
            I8_2;                 // epilog
```

where IN_2 is IN subject to renaming; in our case, only occurrences of t3 are renamed as t3_2 in all IN_2, as seen in the *Instructions After Modulo Variable Expansion* table.

Table 3-13: Instructions After Modulo Variable Expansion

Generic Instruction	Specific Instance
I1 and I1_2	t1=[p1++]
I2 and I2_2	t2=[p2++]
I3	t3=t1+t5
I3_2	t3_2=t1+t5
I4 and I4_2	t4=t2+1
I5	t5=t1+t3
I5_2	t5=t1+t3_2
I6 and I6_2	t6=t4*t5
I7	t7=t6*t3
I7_2	t7=t6*t3_2
I8 and I8_2	[p8++]=t7

Trip Count

Notice that as the modulo scheduler expands the loop kernel to add in the extra variable sets, the iteration count of the generated loop changes from (N-SC) to (N-SC)/MVE. This is because each iteration of the generated loop is now doing more than one iteration of the original loop, so fewer generated iterations are required.

However, this also relies on the compiler knowing that it can divide the loop count in this manner. For example, if the compiler produces a loop with MVE=2 so that the count should be (N-SC)/2, an odd value of (N-SC) causes problems. In these cases, the compiler generates additional *peeled* iterations of the original loop to handle the

remaining iteration. As with rotation, if the compiler cannot determine the value of N, it will make parts of the loop-the kernel or peeled iterations-conditional so that they are executed only for the appropriate values of N.

The number of times the generated loop iterates is called the *trip count*. As explained above, sometimes knowing the trip count is important for efficient scheduling. However, the trip count is not always available.

Lacking it, additional information may be inferred, or passed to the compiler through the loop_count pragma, specifying:

- *Trip modulo*: A number known to divide the trip count

- *Trip minimum*: A lower bound for the trip count

- *Trip maximum*: An upper bound for the trip count

Optimizer Example

The following fractional scalar product loop is used to show how the optimizer works. To see the described behavior, compile the example:

- With the optimizer enabled. For more information, see Using Profile-Guided Optimization.

- With the -sat-associative command-line switch. This switch is required because the example uses fractional operations, which saturate. The compiler does not treat saturating operations as associative, by default, which means they normally prevent vectorization.

Example: C source code for fixed-point scalar product

```
#include <stdfix.h>
#include <builtins.h>
long fract sp(fract *a, fract *b) {
    int i;
    accum sum=0.0k;
    aligned(a, 4);
    aligned(b, 4);
    for (i=0; i<100; i++) {
        sum += a[i] * b[i];
    }
    return (long fract)sum;
}
```

After code generation and conventional scalar optimizations are done, the compiler generates a loop that looks something like the following example:

Example: Initial Code Generated for Fixed-Point Scalar Product

```
    P2 = 100;
    LOOP .P1L3 LC0 = P2;
.P1L3:
    LOOP_BEGIN P1L3;
    R0 = W[P0++] (X);
    R2 = W[P1++] (X);
```

```
    A0 += R0.L * R2.L;
    LOOP_END .P1L3;
.P1L4:
    R0 = A0;
```

The loop exit test has been moved to the bottom and the loop counter rewritten to count down to zero, allowing a zero-overhead loop to be generated. The `sum` is being accumulated in `A0`. `P0` and `P1` are initialized with the parameters `a` and `b`, respectively, and are incremented on each iteration.

To use 32-bit memory accesses, the optimizer unrolls the loop to run two iterations in parallel. The `sum` is now being accumulated in `A0` and `A1`, which must be added together after the loop to produce the final result. To use word loads, the compiler has to know that `P0` and `P1` have initial values that are multiples of four bytes.

This is done in the example by use of `aligned()`, although it could also have been propagated with IPA.

NOTE: Unless the compiler knows that the original loop was executed an even number of times, a conditionally-executed odd iteration must be inserted outside the loop.

NOTE: Vectorization is only possible in this example because the `-sat-associative` switch enables re-ordering of saturating addition and multiplication through associativity. If the example performs an integer scalar product instead of a fractional scalar product, the associativity would be enabled by default.

Example: Code Generated for Fixed-Point Scalar Product After Vectorization Transformation

```
    P2 = 50;
    A1 = A0 = 0;
    LOOP .P1L3 LC0 = P2;
.P1L3:
    LOOP_BEGIN .P1L3;
    R0 = [P0++];
    R2 = [P1++];
    A1+=R0.H*R2.H, A0+=R0.L*R2.L;
    LOOP_END .P1L3;
.P1L4:
    A0 += A1;
    R0 = A0;
```

Finally, the optimizer rotates the loop, unrolling and overlapping iterations to obtain the highest possible use of functional units. Code similar to the following is generated.

Example: Code Generated for Fixed-Point Scalar Product After Software Pipelining

```
    A1=A0=0 || R0 = [P0++] || NOP;
    R2 = [I1++];
    P2 = 49;
    LOOP .P1L3 LC0 = P2;
.P1L3:
    LOOP_BEGIN .P1L3;
    A1+=R0.H*R2.H, A0+=R0.L*R2.L
            || R0 = [P0++]
            || R2 = [I1++];
```

```
   LOOP_END .P1L3;
.P1L4:
   A1+=R0.H*R2.H, A0+=R0.L*R2.L;
   A0 += A1;
   R0 = A0;
```

Assembly Optimizer Annotations

When the compiler optimizations are enabled, the compiler can perform a large number of optimizations to generate the resultant assembly code. The decisions taken by the compiler as to whether certain optimizations are safe or worthwhile are generally invisible to a programmer. However, it can be beneficial to get feedback from the compiler regarding the decisions made during optimization. The intention of the information provided is to give a programmer an understanding of how close to optimal a program is and what more could possibly be done to improve the generated code.

The feedback from the compiler optimizer is provided by means of annotations made to the assembly file generated by the compiler. The assembly file generated by the compiler can be saved by specifying the `-S` switch, the `-save-temps` switch, or by checking the >*Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > General > Save temporary files* option in the IDE.

The assembly code generated by the compiler optimizer is annotated with the following information:

- Global Information

- Procedure Statistics

- Instruction Annotations

- Loop Identification

- Vectorization

- Modulo Scheduling Information

- Warnings, Failure Messages, and Advice

The assembly annotations provide information in several areas that you can use to assist the compiler's evaluation of your source code. In turn, this improves the generated code. For example, annotations could provide indications of resource usage or the absence of a particular optimization from the resultant code. Annotations which note the absence of optimization can often be more important than those noting its presence. Assembly code annotations give the programmer insight into why the compiler enables and disables certain optimizations for a specific code sequence.

Annotation Examples

Your installation directory contains a number of examples which demonstrate the optimizer's annotation output. You can find these examples in the following directory tree:

```
<installation>\Blackfin\Examples\No_HW_Required\ proc \annotations
```

where *proc* is one of:

- `ADSP-BF533` - contains IDE projects pre-configured for the ADSP-BF533 processor.

- `ADSP-BF609` - contains IDE projects pre-configured for the ADSP-BF609 processor.

The examples in this directory tree are not intended to be functional; although they can be built in the IDE and loaded into a processor, they do not *do* anything of significance. Instead, their purpose is to show the kind of annotations generated by the compiler, for a given kind of input source code. In each case, you can import and build the example, as described in Importing Annotation Examples, then examine the resulting assembly file. Depending on the example, you may also see annotations when viewing the C source file in the IDE. Details on how to view the generated annotations is given in:

- Viewing Annotation Examples in the IDE

- Viewing Annotation Examples in Generated Assembly

Importing Annotation Examples

To import an example into the IDE:

1. Choose *File > Import > General.*

2. Select *Existing Projects Into Workspace.*

3. Choose *Select root directory*, and click *Browse.*

4. Navigate to the `Blackfin\Examples\No_HW_Required\`*proc*`\annotations` directory in your installation, for your preferred processor, and click *OK.*

5. The IDE lists the available annotations example projects. Check the examples you want to import.

6. Enalble *Copy projects into Workspace.* This gives you your own working copy of the examples, so that you can build them.

7. Click *Finish.*

NOTE: There is a `Core1` project which can be imported and built for the second core when using the ADSP-BF609 processor examples. This project does not do anything interesting either; it just provides an empty `main()` function pre-configured for loading into Core 1.

Once you have your annotations projects loaded into your IDE, you need to build them. This will produce an executable file. It will also produce generated assembly source files.

NOTE: A lot of diagnostics will appear in the *Console* view when you build any of the annotations examples. This is normal, as annotations are a form of diagnostic, and are emitted to the standard error output as well as to the assembly file.

Viewing Annotation Examples in the IDE

To view the annotations in the IDE:

1. Create a launch configuration for your selected processor, and ensure that the launch configuration loads the executable you built in Importing Annotation Examples. If using the ADSP-BF609 processor, ensure that the configuration loads the executable from the Core1 project into Core 1.

2. Launch the configuration, and let the example run to `main()`.

3. Step into the first function called by `main()`. `main()` itself doesn't do anything interesting.

4. You will see "i" information icons in the left-hand gutter of the source file view. Hover your mouse pointer over these icons to see the annotations that have been associated with the source lines.

5. Alternatively, open the Problems view; annotations are a low-severity form of diagnostic, so are gathered by the Problems view when the application is built.

The annotations examples produce these "i" information icons because they enable annotations diagnostics: if you examine the projects, you will see that they all set *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Warning > Warning/annotation/remark control to Errors, warnings and annotations*.

Viewing Annotation Examples in Generated Assembly

To view annotations in the generated assembly file:

1. Open the annotations project and build it, if you have not already done so.

2. In the Project Explorer view, browse to the `Debugsrc` directory if you built the project using the Debug configuration, or to the `Release\src` directory if you built the project using the Release configuration. You will find several assembly files there (with `.s` suffix).

3. Double-click on the assembly file that corresponds to the example. For example, in the `file_position` example, select `file_example.s`.

4. The IDE will open the assembly file in a source view. You can see the annotations as comments within that generated assembly file.

You can see the generated assembly files because the annotations projects have been configured to have *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > General > Save temporary files* checked. Normally, this setting is off, and the compiler deletes the generated assembly file after it has been converted into an object file.

Global Information

For each compilation unit, the assembly output is annotated with:

- The time of the compilation

- The options used during that compilation.

- The architecture for which the file was compiled.

- The silicon revision used during the compilation

- A summary of the workarounds associated with the specified architecture and silicon revision. These workarounds are divided into:

- Disabled: these are the workarounds that were not applied

- Enabled: these are the workarounds that were applied during the compilation.

- Always on: these are workarounds that are always applied and that cannot be disabled, not even by using the `-si-revision none` compiler switch.

- Never on: these are workarounds that are never applied and that cannot be enabled.

The `global_information` project is an example of this information. Build the project, then open the `hello.s` assembly file. You will see this information at the start of the file.

Procedure Statistics

For each function, the following is reported:

- Frame size - The size of stack frame.

- Registers used - Since function calls tend to implicitly clobber registers, there are several sets:

    - The first set is composed of the scratch registers changed by the current function. This does not count the registers that are implicitly clobbered by the functions called from the current function.

    - The second set are the call-preserved registers changed by the current function. This does not count the registers that are implicitly clobbered by the functions called from the current function.

    - The third set are the registers clobbered by the inner function calls.

- Inlined Functions - If inlining happens, then the header of the caller function reports which functions were inlined inside it and where. Each inlined function is reported using the position of the inlined call. All the functions inlined inside the inlined function are reported as well, generating a tree of inlined calls. Each node, except the root, has this form:

    `file_name:line:column'function_name`

    where:

    - `function_name` is the name of the function inlined.

    - `line` is the line number of the call to `function_name`, in the source file.

    - `column` is the column number of the call to `function_name`, in the source file.

    - `file_name` is the name of the source file calling `function_name`.

The `procedure_statistics` annotation example illustrates this. You can view the annotations in the IDE either via the C source view or the generated assembly.

- In a C source view, the procedure information for each function can be viewed by hovering the mouse pointer over the "i" information icon in the gutter beside the first line of each function declaration-for example, beside `int foo(int in)`, in `procedure_statistics.c`.

- In an assembly source view, the procedure information can be viewed by scrolling down to the label that marks the start of each function-for example, just after the label `_foo:` in `procedure_statistics.s`.

The `procedure_statistics_inlining` demonstrates the annotations produced when a function inlines the contents of another function. Build the project in the Release configuration, and open `Release\src \procedure_statistics_inlining.s`. Observe how calls to functions `f2()` and `f3()` have been inlined into function `f1()`, and how the annotations at label `_f1:` report this.

Note that, if you build using the Debug configuration, you do not see the same annotations, as the optimizer is not enabled, so inlining does not happen.

Instruction Annotations

Sometimes the compiler annotates certain assembly instructions. It does so in order to point to possible inefficiencies in the original source code, or when the -annotate-loop-instr switch is used to annotate the instructions related to modulo-scheduled loops.

The format of an assembly line containing several instructions is changed. Instructions issued in parallel are no longer shown all on the same assembly line; each is shown on a separate assembly line, so that the instruction annotations can be placed after the corresponding instructions. For example,

```
instruction_1, instruction_2, instruction_3;
```

is displayed as:

```
instruction_1, // {annotations for instruction_1}
instruction_2, // {annotations for instruction_2}
instruction_3; // {annotations for instruction_3}
```

Example `instruction_annotations` demonstrates both these kinds of annotation. Build the example using the Release mode.

- When viewing `instruction_annotations.c` in the C source view, you can see that there is an annotation in the `bad_mod()` function to indicate that the division operation is emulated in software. You can also see that the optimizer modulo-scheduled the loop in the `dotprod()` function, but the individual instruction annotations are not available.

- When viewing `instruction_annotations.s` in the assembly source view, you can see the same annotations as for the C source view, but you can also see the additional information for each instruction within the loop in the `dotprod()` function.

Loop Identification

One useful annotation is loop identification-that is, showing the relationship between the source program loops and the generated assembly code. This is not easy due to the various loop optimizations. Some of the original loops may not be present, because they are unrolled. Other loops get merged, making it difficult to describe what has happened to them.

The assembly code generated by the compiler optimizer is annotated with the following loop information:

- Loop Identification Annotations

- Resource Definitions

- File Position

- Infinite Hardware Loop Wrappers

Finally, the assembly code may contain compiler-generated loops that do not correspond to any loop in the user program, but rather represent constructs, such as structure assignment or calls to `memcpy`.

Loop Identification Annotations

Loop identification annotation rules are:

- Annotate only the loops that originate from the C looping constructs `do`, `while`, and `for`. Therefore, any `goto` defined loop is not accounted for.

- A loop is identified by the position of the corresponding keyword (`do`, `while`, `for`) in the source file.

- Account for all such loops in the original user program.

- Generally, loop bodies are delimited between the `Lx: Loop at <file position>` and `End Loop Lx` assembly annotation. The former annotation follows the label of the first block in the loop. The later annotation follows the jump back to the beginning of the loop. However, there are cases in which the code corresponding to a user loop cannot be entirely represented between two markers. In such cases the assembly code contains blocks that belong to a loop, but are not contained between that loop's end markers. Such blocks are annotated with a comment identifying the innermost loop they belong to, `Part of Loop Lx`.

- Sometimes a loop in the original program does not show up in the assembly file because it was either transformed or deleted. In either case, a short description of what happened to the loop is given at the beginning of the function.

  In cases where a loop has been totally deleted (because a source-level loop is never entered), the compiler will issue the following remark (see Warnings, Annotations and Remarks):

  `cc1973: loop never entered - eliminated`

  In cases where a loop control code surrounding a loop body has been removed (because the loop always iterates only once), the compiler will issue the following remark (see Warnings, Annotations and Remarks):

  `cc1974: loop always iterates once - loop converted to linear code`

- A program's innermost loops are those loops that do not contain other loops. In addition to regular loop information, the innermost loops with no control flow and no function calls are annotated with additional information such as:

  - *Cycle count*. The number of cycles needed to execute one iteration of the loop, including the stalls.

- • *Resource usage.* The resources used during one iteration of the loop. For each resource, the compiler shows how many of that resource are used, how many are available and the percentage of utilization during the entire loop. Resources are shown in decreasing order of utilization. Note that 100% utilization means that the corresponding resource is used at its full capacity and represents a bottleneck for the loop.

- • *Register usage.* If the `-annotate-loop-instr` compiler switch is used, then the register usage table is shown. This table has one column for every register that is defined or used inside the loop. The header of the table shows the names of the registers, written on the vertical, top down. The registers that are not accessed do not show up. The columns are grouped on data registers, pointer registers and all other registers. For every cycle in a loop (including stalls) there is a row in the array. The entry for a register has a `*` on that row if the register is either live or being defined at that cycle.

- • *Optimizations.* Some loops are subject to optimizations such as vectorization. These loops receive additional annotations as described in the vectorization section.

- • Sometimes the compiler generates additional loops that may or may not be directly associated with the loops in the user program. Whenever possible, the compiler annotations try to show the relation between such compiler-generated loops and the original source code. For instance, for certain source level loops, the compiler generates two nested loops, with the outer loop behaving as an infinite loop wrapper for the inner loop, and the outer loop is annotated as an infinite wrapper.

The `loop_identification` annotation example shows some of these annotations. Build the example using the Release configuration. The function `bar()` in file `loop_identification.c` contains two loops, written in such a way that the second loop will not be entered: when the first loop completes, the conditions of entry to the second loop are false. When the optimizer is enabled, the compiler can detect this through a process called *constant propagation*, and can delete the second loop entirely.

- • When viewing `loop_identification.c` in a C source window, `i` information icons appear in the gutter next to the lines containing the `for` and `while` keywords that introduce loops. For the first loop, trip count, estimated cycle count and resource usage is given, while for the second loop, the annotation reports that the loop is removed due to constant propagation.

- • When viewing `loop_identification.s` in an assembly source window, an annotation appears following the `_bar:` label, reporting the removed loop. At other points in the function, annotations appear showing that the following code is part of the first loop, or part of the top level of the function (i.e. not in any loop).

Resource Definitions

For each cycle, a Blackfin processor may execute a single 16- or 32-bit instruction, or it may execute a 64-bit multi-issued instruction consisting of a 32-bit instruction and two 16-bit instructions. Blackfin+ processors also support single 64-bit instructions.

In any case, at most one store instruction may be executed. Not all 16-bit instructions are valid for the multi-issue slots, and not all of those may be placed into either slot. Consequently, the resources are divided into group 1 (use of the first 16-bit multi-issue slot) and group 1 or 2 (use of either 16-bit multi-issue slot).

The resource usage is described in terms of missed opportunities by the compiler; in other words, slots where the compiler has had to issue a `NOP` or `MNOP` instruction.

An instruction of the form:

```
R0 = R0 + R1 (NS) || R1 = [P0++] || NOP;
```

has managed to use both the 32-bit ALU slot and one of the 16-bit memory access slots, but has not managed to use the second 16-bit memory access slot. Therefore, this counts as:

- 1 out of 1 possible 32-bit ALU/MAC instructions
- 1 out of 1 possible group 1 instructions
- 1 out of 2 possible group 1 or 2 instructions
- 0 out of 1 possible stores

A single-issued instruction is seen as occupying all issue-slots at once, because the processor cannot issue other instructions in parallel. Consequently, there are no opportunities missed by the compiler. Thus, a single-issue instruction such as:

```
R2 = R0 +  R1;
```

is counted as:

- 1 out of 1 possible 32-bit ALU/MAC instructions
- 1 out of 1 possible group 1 instructions
- 2 out of 2 possible group 1 or 2 instructions
- 1 out of 1 possible stores

This is because the compiler has not had to issue `NOP` instructions or `MNOP` instructions, and so no resources have been unutilized.

The `loop_identification` annotation example shows some of these annotations. Build the example using the Release configuration. The function `bar()` in file `loop_identification.c` contains two loops, written in such a way that the second loop will not be entered: when the first loop completes, the conditions of entry to the second loop are false. When the optimizer is enabled, the compiler can detect this through a process called *constant propagation*, and can delete the second loop entirely.

- When viewing `loop_identification.c` in a C source view, "i" information icons appear in the gutter next to the lines containing the for and while keywords that introduce loops. For the first loop, trip count, estimated cycle count and resource usage is given, while for the second loop, the annotation reports that the loop is removed due to constant propagation.

- When viewing `loop_identification.s` in an assembly source view, an annotation appears following the "`_bar:` label, reporting the removed loop. At other points in the function, annotations appear showing that the following code is part of the first loop, or part of the top level of the function (i.e. not in any loop).

File Position

When the compiler refers to a file position in an annotation, it does so using the file name, line number, and the column number in that file: as `"ExampleC.c" line 4 col 5`.

This scheme uniquely identifies a source code position, unless inlining is involved. In the presence of inlining, a piece of code from a certain file position can be inlined at several places, which in turn can be inlined at other places. Since inlining can happen an unspecified number of times, a recursive scheme is used to describe a general file position.

Therefore, a `<general file position>` is `<file position>` inlined from `<general file position>`.

Annotations example `file_position` demonstrates this. When built using the Release configuration, two levels of inlining occur in file `file_position.c`:

- When viewing `file_position.c` in a C source view, the loop at the start of function `f3()` has an "i" information indicating that the loop has been inlined into function `f2()` twice, and that each of those instances have in turn been inlined into function `f1()`.

- When viewing `file_position.s` in an assembly source view, annotations appear in the generated file immediately before the code for the loop. The annotations in function `f2()` indicate that the following code was inlined from function `f3()`, and the annotations in function `f1()` indicate that the following code was inlined from function `f2()`, which in turn was inlined from function `f3()`. There are also annotations at the start of functions `f2()` and `f1()` reporting which functions have been inlined into them, as described in Procedure Statistics.

Infinite Hardware Loop Wrappers

The compiler tries to generate hardware loops whenever possible to avoid the delays involved with jump instructions. But hardware loops require a trip count, and that is not always available. For instance, consider this loop whose exit condition is not given by a trip count:

do { body } while (condition);

The compiler could generate code like this:
```
L_start:
   body;
   CC = condition;
   IF CC JUMP L_start (bp);
```

This way the conditional jump takes at least 5 cycles during each iteration. However, if we had a hardware loop that could run forever, then the following alternative would be better:

LOOP L_start LC0 = infinite; LOOP_BEGIN L_start; body; CC = condition; IF !CC JUMP L_out; LOOP_END L_start; L_out:

This is 4 cycles better as the conditional jump takes only one cycle if it is not taken. However, the hardware does not have infinite hardware loops, so the compiler emulates them by using the highest possible trip count for the hardware loop, and wrapping the loop in an infinite loop:

```
L_infinite_wrapper:
P0 = -1;
LOOP L_start LC0 = P0;
LOOP_BEGIN L_start;
   body;
   CC = condition;
   IF !CC JUMP L_out;
LOOP_END L_start;
   JUMP L_infinite_wrapper;
// end loop infinite_wrapper
L_out:
```

The two loops behave as a single infinite loop, with a minor overhead, even though the hardware loop has to terminate. If the condition is never satisfied, the outer loop is executed forever.

The compiler annotations annotate the outer loop as the infinite hardware loop wrapper for the inner loop.

The `hardware_loop_wrappers` annotation example demonstrates this. The function `pseudo_mod()` in file `hardware_loop_wrappers.c` contains a loop of indeterminate count. When built using the Release configuration, the compiler will generate a hardware loop with an outer wrapper.

- When viewing `hardware_loop_wrappers.c` in a C source view, there is an "i" information icon next to the loop in function `pseudo_mod()`. The corresponding annotations include one which reports it is an infinite hardware loop wrapper.

- When viewing `hardware_loop_wrappers.s` in an assembly source view, there are several annotations for the loop in function `pseudo_mod()`. The first one indicates that it is the infinite hardware loop wrapper.

Vectorization

The trip count of a loop is the number of times the loop body gets executed.

Under certain conditions, the compiler can take two operations from consecutive iterations of a loop and execute them in a single, more powerful instruction. This gives a loop a smaller trip count. The transformation in which operations from two subsequent iterations are executed in one more powerful single operation is called "*vectorization*.

For instance, the original loop may start with a trip count of 1000.

```
for(i=0; i < 1000; ++i)
   a[i] = b[i] + c[i];
```

After the optimization, the vectorized loop has a final trip count of 500. The vectorization factor is the number of operations in the original loop that are executed at once in the transformed loop. It is illustrated using some pseudo code below.

```
for(i=0; i < 1000; i+=2)
   (a[i], a[i+1]) = (b[i],b[i+1]) .plus2. (c[i], c[i+1])
```

In the above example, the vectorization factor is 2. A loop may be vectorized more than once.

If the trip count is not a multiple of the vectorization factor, some iterations need to be peeled off and executed unvectorized. If in the previous example, the trip count of the original loop was 1001, then the vectorized code would be:

```
for(i=0; i < 1000; i+=2)
   (a[i], a[i+1]) = (b[i],b[i+1]) .plus2. (c[i], c[i+1]);
    a[1000] = b[1000] + c[1000];
                // This is one iteration peeled from
                // the back of the loop.
```

In the above examples, the trip count is known and the amount of peeling is also known. If the trip count (a variable) is not known, the number of peeled iterations depends on the trip count. In such cases, the optimized code contains peeled iterations that are executed conditionally.

Unroll and Jam

A vectorization-related transformation is unroll and jam. Where the source file has two nested loops, sometimes the compiler can unroll the outer loop, to create two copies of the inner loop each operate on different iterations of the loop. It can then "jam" these two loops together, interleaving their operations, giving a sequence of operations that is more amenable to vectorization. The compiler issues annotations when this transformation has happened.

The `unroll_and_jam` annotation example demonstrates this. The example contains three source files:

- `unroll_and_jam_original.c` - the "real" example. This file contains a function which the compiler is able to optimize using the unroll-and-jam transformation.

- `unroll_and_jam_unrolled.c` - this file is illustrative of how the compiler's internal representation would be, part-way through the unroll-and-jam transformation. This is not an example of how you should write your code. In this representation, the compiler has unrolled the outer loop once, so that there are two complete, separate copies of the inner loop. The first copy works on even iterations, while the second works on odd iterations.

- `unroll_and_jam_jammed.c` - another illustrative representation of the function, after the transformation is complete. The compiler has taken the two copies of the loop and overlapped them, then vectorized the operations so that the 16-bit loads and stores are now 32-bit loads and stores that access two adjacent locations in parallel, and the accumulation operations do two separate 16-bit additions in the same cycle.

NOTE: You should always write your code in the cleanest manner possible, to most clearly express your intention to the compiler. You should not attempt to apply transformations such as unroll-and-jam explicitly within your code, as that will obscure your intent and inhibit the optimizer. The unrolled and jammed files are only presented here to illustrate the behavior of the transformation.

The `unroll_and_jam` annotation example makes use of the `unroll_and_jam_original.c` file to demonstrate the annotation produced during this transformation. Build the example using the Release configuration.

- When viewing the `unroll_and_jam_original.c` file in a C source view, there is an "i" information icon next to the outer loop, reporting that the loop has been unrolled and jammed.

- When viewing the `unroll_and_jam_original.s` file in an assembly source view, there is an annotation preceding the generated code for the outer loop, reporting that the loop has been unrolled and jammed.

Loop Flattening

Another transformation, related to vectorization, is *loop flattening*. Loop flattening takes two nested loops that run `N1` and `N2` times respectively, and transforms them into a single loop that runs `N1*N2` times.

The `loop_flattening` annotation example demonstrates this. It contains two files to illustrate the transformation:

- `loop_flattening_original.c` - This file contains two nested loops, iterating 30 times and 100 times, respectively.

- `loop_flattening_flattened.c` - This file contains a single loop, iterating 3000 times. This file is not an example of how you should write your code-it is merely an illustration of the transformation applied by the compiler optimizer.

The `loop_flattening` annotation example uses `loop_flattening_original.c` to demonstrate the annotations produced. Build the example using the Release configuration.

- When viewing `loop_flattening_original.c` in a C source view, there is an annotation on the outer loop, indicating that the two loops were flattened into one.

- When viewing `loop_flattening_original.s` in an assembly source view, there is an annotation at the beginning of the function, indicating that the two loops were flattened into one; the annotation appears at the start of the function because a loop was "lost" (the loop's structure was removed), and lost loops are reported at the start of each function.

Vectorization Annotations

For every loop that is vectorized, the following information is provided:

- The vectorization factor

- The number of peeled iterations

- The position of the peeled iterations (front or back of the loop)

- Information about whether peeled iterations are conditionally or unconditionally executed

For every loop pair subject to unroll and jam, the following information is provided:

- The number of times the unrolled outer loop was unrolled

- The number of times the inner loop was jammed

For every loop pair subject to loop flattening, the following information is provided:

- The loop that is lost

- The remaining loop that it was merged with

The `vectorization` annotation example demonstrates some of this. File `vectorization.c` contains a function `copy()` which the compiler can conditionally vectorize, when optimizing. Build the example using the Release configuration.

- When viewing `vectorization.c` in a C source view, there are "i" information icons next to the loop constructs in the `copy()` function. These annotations report that there are multiple versions of the loop, one of which is unvectorized; that a loop was vectorized by a factor of two; the trip counts for the loops; and so on.

- When viewing `vectorization.s` in an assembly source view, there are multiple versions of the loop in the function. One has annotations to indicate it has been vectorized, while the other has an annotation to indicate that it is the unvectorized version of the same loop.

Modulo Scheduling Information

For every modulo-scheduled loop (see also Modulo Scheduling), in addition to regular loop annotations, the following information is provided:

- The initiation interval (II)

- The final trip count if it is known: the trip count of the loop as it ends up in the assembly code

- A cycle count representing the time to run one iteration of the pipelined loop

- The minimum trip count, if it is known and the trip count is unknown

- The maximum trip count, if it is known and the trip count is unknown

- The trip modulo, if it is known and the trip count is unknown

- The stage count (iterations in parallel)

- The MVE unroll factor

- The resource usage

- The minimum initiation interval due to resources (`res MII`)

- The minimum initiation interval due to dependency cycles (`rec MII`)

Annotations for Modulo-Scheduled Instructions

The `-annotate-loop-instr` switch can be used to produce additional annotation information for the instructions that belong to the prolog, kernel, or epilog of the modulo-scheduled loop.

Consider the example whose schedule is in the ***Modulo Schedule Corrected*** table (see Variable Expansion and MVE Unroll). Remember that this example does not use a real DSP architecture, but rather a theoretical one able to schedule four instructions on a line, and each line takes one cycle to execute. We can view the instructions involved in modulo scheduling as in the ***Modulo-Scheduled Instructions*** table.

Due to variable expansion, the body of the modulo-scheduled loop contains `MVE=2` unrolled instances of the kernel, and the loop body contains instructions from 4 iterations of the original loop. The iterations in progress in the kernel are shown in the table heading, starting with `Iteration 0`, which is the oldest iteration in progress (in its final stage). This example uses two register sets, shown in the table heading.

Table 3-14: Modulo-Scheduled Instructions

	Part	Iteration 0	Iteration 1	Iteration 2	Iteration 3 ...
		Register Set 0	Register Set 1	Register Set 0	Register Set 1
1	prolog	I1			
2	prolog	I2, I3			
3	prolog	I4, I5	I1_2		
4	prolog	I6	I2_2, I3_2		
5		L: Loop ...			
6	*kernel*	*I7*	*I4_2, I5_2*	*I1*	
7	*kernel*	*I8*	*I6_2*	*I2, I3*	
8	*kernel*		*I7_2*	*I4, I5*	*I1_2*
9	*kernel*		*I8_2*	*I6*	*I2_2, I3_2*
10		END Loop			
11	epilog			I7	I4_2, I5_2
12	epilog			I8	I6_2
13	epilog				I7_2
14	epilog				I8_2

The instruction annotations contain the following information:

- The part of the modulo-scheduled loop (prolog, kernel, or epilog)

- The loop label: This is required since prolog and epilog instructions appear outside of the loop body and are subject to being scheduled with other instructions.

- ID: A unique number associated with the original instruction in the unscheduled loop that generates the current instruction. It is useful because a single instruction in the original loop can expand into multiple instructions in a modulo-scheduled loop.

  In our example, the annotations for all instances of `I1` and `I1_2` have the same ID, meaning they all originate from the same instruction (`I1`) in the unscheduled loop. The IDs are assigned in the order the instructions appear in the kernel and they might repeat for MVE unroll > 1.

- Loop-carry path, if any: If an instruction belongs to the loop-carry path, its annotation contains a `*`. If several such paths exist, `*2` is used for the second one, `*3` for the third one, and so on.

- `sn`: The stage count to which the instruction belongs

- `rs`: The register set used for the current instruction (useful when MVE unroll > 1, in which case `rs` can be `0`, `1`, ..., `mve-1`). If the loop has an MVE of 1, the instruction's `rs` is not shown.

- Additionally, the instructions in the kernel are annotated with:

  - Iteration. `Iter`: specifies the iteration of the original loop an instruction is on in the schedule.

  - In a modulo-scheduled kernel, there are instructions from (`SC+MVE-1`) iterations of the original loop. `Iter=0` denotes instructions from the earliest iteration of the original loop, with higher numbers denoting later iterations.

Thus, the instructions corresponding to the schedule in the *Modulo-Scheduled Instructions* table for a hypothetical machine are annotated as follows:

```
1 :  I1;    // {L10 prolog:id=1,sn=0,rs=0}
2 :  I2,    // {L10 prolog:id=2,sn=0,rs=0}
3 :  I3;    // {L10 prolog:id=3,sn=0,rs=0}
4 :  I4,    // {L10 prolog:id=4,sn=1,rs=0}
5 :  I5,    // {L10 prolog:id=5,sn=1,rs=0}
6 :  I1_2;  // {L10 prolog:id=1,sn=0,rs=1}
7 :  I6,    // {L10 prolog:id=6,sn=1,rs=0}
8 :  I2_2,  // {L10 prolog:id=2,sn=0,rs=1}
9 :  I3_2;  // {L10 prolog:id=3,sn=0,rs=1}
10://-----------------------------------------------------
11://  Loop at ...
12://-----------------------------------------------------
13://  This loop executes 2 iterations of the original loop
    //  in estimated 4 cycles.
14://-----------------------------------------------------
15://  Unknown Trip Count
16://  Successfully found modulo schedule with:
17://  Initiation Interval (II) = 2
18://  Stage Count (SC) = 3
19://  MVE Unroll Factor = 2
20://  Minimum initiation interval due to recurrences
    //  (rec MII) = 2
21://  Minimum initiation interval due to resources
    //  (res MII) = 2.00
22://-----------------------------------------------------
23:L10:
23:LOOP (N-2)/2;
25:  I7,   // {kernel:id=7,sn=2,rs=0,iter=0}
26:  I4_2,// {kernel:id=4,sn=1,rs=1,iter=1}
27:  I5_2,// {kernel:id=5,sn=1,rs=1,iter=1,*}
28:  I1;   // {kernel:id=1,sn=0,rs=0,iter=2}
29:  I8,   // {kernel:id=8,sn=2,rs=0,iter=0}
30:  I6_2,// {kernel:id=6,sn=1,rs=1,iter=1}
31:  I2,   // {kernel:id=2,sn=0,rs=0,iter=2}
32:  I3;   // {kernel:id=3,sn=0,rs=0,iter=2,*}
```

```
33:  I7_2,// {kernel:id=7,sn=2,rs=1,iter=1}
34:  I4,  // {kernel:id=4,sn=1,rs=0,iter=2}
35:  I5,  // {kernel:id=5,sn=1,rs=0,iter=2,*}
36:  I1_2;// {kernel:id=1,sn=0,rs=1,iter=3}
37:  I8_2,// {kernel:id=8,sn=2,rs=1,iter=1}
38:  I6,  // {kernel:id=6,sn=1,rs=0,iter=2}
39:  I2_2,// {kernel:id=2,sn=0,rs=1,iter=3}
40:  I3_2;// {kernel:id=3,sn=0,rs=1,iter=3,*}
41:  END LOOP
42:
43:  I7,  // {L10 epilog:id=7,sn=2,rs=0}
44:  I4_2,// {L10 epilog:id=4,sn=1,rs=1}
45:  I5_2;// {L10 epilog:id=5,sn=1,rs=1}
46:  I8,  // {L10 epilog:id=8,sn=2,rs=0}
47:  I6_2;// {L10 epilog:id=6,sn=1,rs=1}
48:  I7_2;// {L10 epilog:id=7,sn=2,rs=1}
49:  I8_2;// {L10 epilog:id=8,sn=2,rs=1}
```

Lines 10-22 define the kernel information: loop name and modulo-schedule parameters: II, stage count, etc.

Lines 25-40 show the kernel.

Each instruction in the kernel has an annotation between { }, inside a comment following the instruction. If several instructions are executed in parallel, each gets its own annotation.

For instance, line 27 looks like:

```
27:  I5_2,  // {kernel:id=5,sn=1,rs=1,iter=1,*}
```

This annotation describes:

- That this instruction belongs to the kernel of the loop starting at `L10`.

- That this and the other three instructions that have `ID=5` originate from the same original instruction in the unscheduled loop:

```
5:  I5,     // {L10 prolog:id=5,sn=1,rs=0}
...
27:  I5_2, // {kernel:id=5,sn=1,rs=1,iter=1,*}
...
35:  I5,    // {kernel:id=5,sn=1,rs=0,iter=2,*}
...
45:  I5_2; // {L10 epilog:id=5,sn=1,rs=1}
```

- `sn=1` shows that this instruction belongs to stage count 1.

- `rs=1` shows that this instruction uses register set 1.

- `Iter=1` specifies that this instruction belongs to the second iteration of the original loop (`Iter` numbers are zero-based).

- The `*` indicates that this is part of a loop carry path for the loop. In the original, unscheduled loop, that path is I5 -> I3 -> I5. Due to unrolling, in the scheduled loop the "unrolled" path is I5_2-> I3->I5->I3_2->I5_2.

The prolog and epilog are not clearly delimited in blocks by themselves, but their corresponding instructions are annotated like the ones in the kernel except that they do not have an `Iter` field and that they are preceded by a tag specifying which prolog or epilog they belong to:

```
5 : I5,   // {L10 prolog:id=5,sn=1,rs=0}
...
27: I5_2, // {kernel:id=5,sn=1,rs=1,iter=1,*}
...
35: I5,   // {kernel:id=5,sn=1,rs=0,iter=2,*}
...
5: I5_2;  // {L10 epilog:id=5,sn=1,rs=1}
```

Note that the prolog/epilog instructions may mix with other instructions on the same line.

This situation does not occur in this example; however, in a different example it might have:

```
  I5_2,     // L10 epilog:id=5,sn=1,rs=1} I20;
```

This shows a line with two instructions. The second instruction `I20` is unrelated to modulo scheduling, and therefore it has no annotation.

Warnings, Failure Messages, and Advice

There are innocuous programming constructs that have a negative effect on performance. Since you may not be aware of the hidden problems, the compiler annotations try to give warnings when such situations occur. Also, if a program construct keeps the compiler from performing a certain optimization, the compiler gives the reason why that optimization was precluded.

In some cases, the compiler assumes it could do a better job if you changed your code in certain ways. In these cases, the compiler offers advice on the potentially beneficial code changes. However, take this cautiously. While it is likely that making the suggested change will improve the performance, there is no guarantee that it will actually do so.

Some of the messages are:

- *This loop was not modulo scheduled because it was optimized for space* When a loop is modulo-scheduled, it often produces code that has to precede the scheduled loop (the prolog) and follow the scheduled loop (the epilog). This almost always increases the size of the code. That is why, if you specify an optimization that minimizes the space requirements, the compiler doesn't attempt modulo scheduling of a loop.

- *This loop was not modulo scheduled because it contains calls or volatile operations* Due to the restrictions imposed by calls and volatile memory accesses, the compiler does not try to modulo-schedule loops containing such instructions.

- *This loop was not modulo scheduled because it contains too many instructions* The compiler does not try to modulo-schedule loops that contain many instructions, because the potential for gain is not worth the increased compilation time.

- *This loop was not modulo scheduled because it contains jump instructions* Only single block loops are modulo-scheduled. You can attempt to restructure your code and use single block loops.

- *This loop would vectorize more if alignment were known* The loop was vectorized, but it could be vectorized even more if the compiler could deduce a stronger alignment of some memory locations used in the loop.

- *This loop would vectorize if alignment were known* The loop was not vectorized because of unknown pointer alignment.

- *Consider using pragma loop_count to specify the trip count or trip modulo* This information may help vectorization.

- *Consider using pragma loop_count to specify the trip count or trip modulo, in order to prevent peeling* When a loop is vectorized, but the trip count is not known, some iterations are peeled from the loop and executed conditionally (based on the run-time value of the trip count). This can be avoided if the trip count is known to be divisible by the number of iterations executed in parallel as a result of vectorization.

- *operation of this size is implemented as a library call* This message is issued when source code operator operation results in a library call, due to lack of hardware support for performing that operation on operands of that size.

  In this case the compiler will also issue the following remark (see Warnings, Annotations and Remarks):

  `cc2261`: *operation* implemented as a library call

- operation is implemented as a *library call* This message is issued when source code operator *operation* results in a library call, due to lack of direct hardware support. For instance, an integer division results in a library call. In this case the compiler will also issue the following remark (see Warnings, Annotations and Remarks):

  `cc2261`: *operation* implemented as a library call

- *MIN operation could not be generated because of unsigned operands* This message is issued when the compiler detects a MIN operation performed between unsigned values. Such an operation cannot be implemented using the hardware `MIN` instruction, which requires signed values.

- *MAX operation could not be generated because of unsigned operands* This message is issued when the compiler detects a MAX operation performed between unsigned values. Such an operation cannot be implemented using the hardware `MAX` instruction, which requires signed values.

- *Use of volatile in loops precludes optimizations* In general, volatile variables hinder optimizations. They cannot be promoted to registers, because each access to a volatile variable requires accessing the corresponding memory location. The negative effect on performance is amplified if volatile variables are used inside loops. However, there are legitimate cases when you have to use a volatile variable exactly because of this special treatment by the optimizer. One example would be a loop polling if a certain asynchronous condition occurs. This message does not discourage the use of volatile variables, it just stresses the implications of such a decision.

- *Jumps out of this loop prevent efficient hardware loop generation* Due to the presence of jumps out of a loop, the compiler either cannot generate a hardware loop, or was forced to generate one that has a conditional exit.

- *Consider using a 4-byte integral type for the variable name, for more efficient hardware loop generation* Using short-typed variables as loop control variables limits optimization because the short variables may wrap. For instance, in the following example:

```
unsigned short i;
for (i = 0; i < c; i++)
    ....
```

if `c` > 65536, then the loop will run forever because `i` wraps from 65535 back to 0. The compiler recommends using an `int` variable instead (`int` or `unsigned int`) unless the smaller size is critical to your program's behavior.

- *There are N more instructions related to this call* Certain operations are implemented as library calls. In those cases the call instruction in the assembly code is annotated explaining that the user operation was implemented as a call. However the cost of the operation may be slightly larger than the cost of the call itself, due to additional overhead required to pass the parameters and to obtain the result. This message gives an estimate of the number of instructions in such an overhead associated with a library call.

- *This function calls the "alloca" function which may increase the frame size* The assembly annotations try to estimate the frame size for a given function. However, if the function makes explicit use of `alloca` then this increases the frame size beyond the original reported estimate.

Analyzing Your Application

The compiler and run-time libraries provide several features for analyzing the run-time behavior of your application. These features allow you to better debug errors and fine-tune the program. Features discussed in this chapter are:

- Application Analysis Configuration discusses general control of the analysis features.

- Profiling With Instrumented Code discusses how to profile the application, measuring the time spent in individual functions in an application.

- Profile-Guided Optimization and Code Coverage discusses how to improve application performance using profile guided optimization (PGO). Producing code coverage reports using profile guided optimization data is also discussed.

- Heap Debugging details how to use the run-time library heap debugging feature to identify heap-allocated memory leaks and heap-allocated memory corruption within an application.

- Stack Overflow Detection details how to use the stack overflow feature to determine when an application has exceeded its maximum stack size.

Application Analysis Configuration

The analysis features described in this section can be configured through some global settings, used by an underlying profiling layer. The `<sys\adi_prof.h>` header file exposes this layer. The following aspects can be controlled:

- Application Analysis and File Naming

- Device for Profiling Output

- Frequency of Flushing Profile Data

Application Analysis and File Naming

The analysis features, described in this section, each rely on files created by the application while it is running. In order for the analysis tools to be able to locate such files, the application and tools must agree on the file names. This agreement is achieved through the EXECUTABLE_NAME linker directive, which allows an application to discover the name of its own executable image. The run-time library can then use this name as the basis of the generated files, thus tying the generated file to the executable that created it. This method allows the Reporter Tool to produce useful reports based on the application and its generated log files.

The features that use this functionality are:

- Profile-guided optimization (PGO) for hardware

- Instrumented profiling

- Heap debugging

The EXECUTABLE_NAME directive takes an assembler symbol name as a parameter. For the features in this chapter, the symbol name must be __executable_name.

> **NOTE:** You do not need to add the __executable_name symbol to your application. The linker automatically creates an object file containing the declaration of the symbol when it encounters the EXECUTABLE_NAME directive in the .ldf file.

The __executable_name assembler symbol declared in the .ldf file can be referenced in C/C++ applications. The data is stored in a NUL-terminated C string.

As an alternative to using the EXECUTABLE_NAME directive, you can provide a declaration of the symbol within your application, for example, in C:

```
char _executable_name[] = "my_executable.dxe";
```

> **NOTE:** If no EXECUTABLE_NAME directive is provided in the .ldf file, the application reverts to using the default definition of __executable_name. This definition contains the string unknown.dxe.

Device for Profiling Output

The profiling features require an underlying I/O device driver to produce output to either stderr or the appropriate log file. The features will use the device driver specified by the integer adi_prof_io_device. If adi_prof_io_device is -1, the profilers will use the default device driver. adi_prof_io_device defaults to -1, but this definition can be overridden with a value representing the required device driver.

Frequency of Flushing Profile Data

To reduce the impact of I/O operations, the profilers buffer data internally, and write the data to the log files in bursts. The intervals can be controlled through the following global variables:

- adi_prof_min_flush_interval determines the minimum time that must pass between buffer flushes.

- `adi_prof_max_flush_interval` determines the maximum time that may pass between buffer flushes. This value is used to determine whether to flush data to the log file before the buffer fills.

The library provides default values for each of these variables, but you can override the defaults just by defining your own versions. For example:

```
uint32_t adi_prof_max_flush_interval
            = ADI_MSEC_FLUSH_INTERVAL(10000); // 10 seconds
```

NOTE: The `ADI_MSEC_FLUSH_INTERVAL` macro is based upon the `__PROCESSOR_SPEED__` macro.

Profiling With Instrumented Code

Instrumented profiling is an application profiling tool that provides a summary of cycle counts for functions within an application. To produce an instrumented profiling summary:

1. Ensure the *C/C++ I/O device support* option is enabled in the *Startup Code Configuration* dialog box. If *C/C++ I/O device support* is disabled, the instrumented profiling cannot open its output file and issues an error.

2. Compile your application with the -p switch, or with *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor > Enable compiler instrumented profiling* selected. For best results, use the optimization switches that will be enabled in the released version of the application.

3. Gather the profile. Run the executable with a training data set. The training data set should be representative of the data that you expect the application to process in the field. The profile is stored in a file with the extension `.prf`.

4. Generate the profiling report. Two options for creating reports are available:

   a. Using the IDE; this will produce an HTML format report.

   b. Using the command-line tools; this will produce a plain-text report.

5. Based on the profiling report, modify the application to improve performance in critical sections of code.

NOTE: Instrumented profiling works by planting function calls into your application which record the cycle count (and in multi-threaded cases, the thread identifier) at certain points. Applications built with instrumented profiling should be used for development and should not be released.

NOTE: Instrumented profiling requires that an I/O device is available in the application to produce its profiling data. The default I/O device will be used to perform I/O operations for instrumented profiling.

NOTE: Instrumented profiling flushes any remaining profile data still pending when `exit()` is invoked. Multi-threaded applications may need to flush data explicitly.

Generating an Application With Instrumented Profiling

The -p compiler switch enables instrumented profiling in the compiler when compiling C/C++ source into assembly. The compiler cannot instrument assembly files or files that have already been compiled into object files.

You can enable the −p switch in an IDE project via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor > Enable compiler instrumented profiling*.

> **NOTE:** When compiling with the −p switch, the compiler and linker will define the preprocessor macro `_INSTRUMENTED_PROFILING` with a value of 1.

Running the Executable

To produce a profiling report, run the application in either the simulator or on hardware. The application will produce a profiling file which is used to create the profiling report. The profiling file will be located in the same directory as the executable, and named as per the executable with a `.prf` suffix.

> **NOTE:** If the application's `.ldf` file does not use the `EXECUTABLE_NAME` directive, the profiling file will revert to the legacy name of unknown.prf. For more information, see Application Analysis and File Naming.

The profiling output file needs to be converted into a readable report. This can be achieved using one of two tools: the IDE Reporter Tool or the command-line `instrprof.exe` tool. See Generating an Instrumented Profiling Report and Invoking the instrprof.exe Command-Line Reporter for information on how to produce a report from the `.prf` profile data file.

Generating an Instrumented Profiling Report

The default output format for the Instrumented Profiling report is pre-formatted HTML. To produce the HTML file:

1. Select *File > New > Code Analysis Report*.

2. Select *Instrumented profiling*.

3. Enter the name of the application executable in the *DXE that produced the data* field.

4. If the application `.ldf` file does not contain an `EXECUTABLE_NAME` directive, the *Data file* field will not have been automatically updated. Enter the name of the `.prf` profiling data file into the field.

5. Enter the filename for the HTML report that will be generated.

6. Click *Finish*.

To generate a report from the command line, in either pre-formatted HTML or unformatted XML, refer to the Reporter.exe Command Line Report Generation Utility.

Invoking the instrprof.exe Command-Line Reporter

The `instrprof.exe` command-line tool produces a plain-text report printed to the command-line console. To produce a report, invoke the `instrprof.exe` tool, providing the application executable and the `.prf` profiling data file as parameters. For example,

```
instrprof.exe test.dxe test.prf
```

The report is displayed via standard output, typically to the console.

Contents of the Profiling Report

The profiling report lists each profiled function called in the application, how many times it was called, and cycle counts for that function. In multi-threaded applications, the thread identifier is also displayed. The Reporter Tool and `instrprof` command-line program present the same information, but in different formats according to their output media. The respective formats are described in Reporter Tool Report Format and instrprof Command-Line Tool Report Format.

Example Program for Instrumented Profiling

```
int apples, bananas;

void apple(void) {
   apples++;    // 10 cycles
}

void banana(void) {
   bananas++;   // 10 cycles
   apple();     // 10 cycles
}  // 20 cycles

int main(void) {
   apple();     // 10 cycles
   apple();     // 10 cycles
   banana();    // 20 cycles
   return 0;    // 40 inclusive cycles total
}  // + exclusive cycles for main itself
```

For example, in the program shown in *Example Program for Instrumented Profiling*, assume that `apple()` takes 10 cycles per call and assume that `banana()` takes 20 cycles per call, of which 10 are accounted for by its call to `apple()`. The program, when run, calls `apple()` three times: twice directly and once indirectly through `banana()`. The `apple()` function clocks up 30 cycles of execution, and this is reported for both its inclusive and exclusive times, since `apple()` does not call other functions. The `banana()` function is called only once. It reports 10 cycles for its exclusive time, and 20 cycles for its inclusive time. The exclusive cycles are for the time when `banana()` is incrementing bananas and is not "waiting" for another function to return, and so it reports 10 cycles. The inclusive cycles include these 10 exclusive cycles and also include the 10 cycles `apple()` used when called from `banana()`, giving a total of 20 inclusive cycles.

The `main()` function is called only once, and calls three other functions (`apple()` twice, `banana()` once). Between them, `apple()` and `banana()` use up to 40 cycles, which appear in the `main()` function's inclusive cycles. The `main()` function's exclusive cycles are for the time when `main()` is running, but is not in the middle of a call to either `apple()` or `banana()`.

NOTE: Time spent in unprofiled functions will be added to the exclusive cycle count for the innermost profiled function, if one is active. (An active profiled function is a profiled function which has an entry in the call stack, that is, it has begun execution but has not yet returned.) For example, if `apple()` called the system function `malloc()`, the time spent in `malloc()` (which is uninstrumented) will be added to the time for `apple()`.

Reporter Tool Report Format

The HTML-formatted instrumented profiling report, produced by the IDE's Reporter Tool, contains a summary of information for the application. Each profiled function called during execution is listed with the following information:

- The function's name

- The pathname of the source file containing the function

- The number of times this function was called

- "Number of cycles without calls": the total number of cycles spent executing the code of this function; if the function calls other profiled functions, the cycles spent in those functions is not included in this figure. Note that if the function calls other *non*-profiled functions, this figure *will* include the cycles spent in those functions.

- "Number of cycles with calls": the total number of cycles spent executing this function, or any function it calls. In other words, this figure gives the sum of cycle counts between this function being called, and it returning.

- The percentage of time spent in this function. This percentage is based on the "number of cycles without calls."

- The thread identifier, for a multi-threaded application.

instrprof Command-Line Tool Report Format

The `instrprof.exe` tool emits a report to standard output. The following is an example of the instrprof output:

```
Summary for thread 1
Function Name        ExecCount        Fn Only        Fn+nested
        main                 1             40               80
       apple                 3             30               30
      banana                 1             10               20
Functional Summary:
Function Name        ExecCount        Fn Only        Fn+nested
        main                 1             40               80
       apple                 3             30               30
      banana                 1             10               20
```

This report includes the following information, for each profiled function:

- The function's name.

- `ExecCount`: the number of times this function was called.

- `Fn Only`: this is the same value as "Number of cycles without calls", as described in Reporter Tool Report Format.

- `Fn+nested`: this is the same value as "Number of cycles with calls", as described in Reporter Tool Report Format.

The report gives a breakdown for each thread in the application, plus an overall combined report for all threads. In this single-threaded example, there is only one thread, so both portions of the report contain the same information.

Profiling Data Storage

The profiling information is stored at runtime in memory allocated from the system heap. If the profiling run-time support cannot allocate from the heap (usually because the heap is exhausted), the profiling runtime calls adi_fatal_error and stops execution. The profiling data available when this happens will be incomplete and probably not very useful. To avoid this problem, increase the size of the system heap until the error is no longer seen when running. For more information, see Controlling System Heap Size and Placement in the Compiler chapter.

> NOTE: Although instrumented profiling uses the default heap for some of its internal storage, none of these allocations will appear in a heap usage report.

Computing Cycle Counts

When profiling is enabled, the compiler instruments the generated code by inserting calls to a profiling library at the start of and end of each compiled function. The profiling library samples the processor's cycle counter and records this figure against the function just started or just completed. The profiling library itself consumes some cycles, and these overheads are not included in the figures reported for each function, so the total cycles reported for the application by the profiler will be less than the cycles consumed during the life of the application. In addition to this overhead, there is some approximation involved in sampling the cycle counter, because the profiler cannot guarantee how many cycles will pass between a function's first instruction and the sample. This is affected by the optimization levels, the state preserved by the function, and the contents of the processor's pipeline. The profiling library knows how long the call entry and exit takes "on average", and adjusts its counts accordingly. Because of this adjustment, profiling using instrumented code provides an approximate figure, with a small margin for error. This margin is more significant for functions with a small number of instructions than for functions with a large number of instructions.

Multi-Threaded and Non-Terminating Applications

When an instrumented application is executed, it records data in the application, occasionally flushing this data to the host computer. In multi-threaded applications and non-terminating single-threaded applications, a request to flush data is required to ensure that all the profiling data is flushed from the application.

> NOTE: In multi-threaded projects, the default thread stack size may not be sufficient for profiling some applications, and may result in unexpected run-time behavior. Refer to your RTOS documentation for instructions on increasing your thread stack size.

Flushing Profile Data

To flush profiling data, the application must include the header file `instrprof.h` and call the function `instrprof_request_flush()`. Any changes to the code for instrumented profiling can be guarded by the preprocessor macro `_INSTRUMENTED_PROFILING`. For example:

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

```
void myfunc_noreturn(int x) {
   while ( 1 ) {
      // Perform operation
    #if defined(_INSTRUMENTED_PROFILING)
      instrprof_request_flush();
    #endif
   }
}
```

The flush will occur when the call to `instrprof_request_flush()` is made. Flushing cannot occur when the scheduler is disabled or from within interrupt handlers.

Profiling of Interrupts and Kernel Time

A single-threaded application (that is, not built with the `-threads` compiler switch) will add any time spent in interrupts to the time of the innermost, active profiled function that was interrupted. Time spent in the interrupt handler will not be visible in the profiling report produced. The compiler does not instrument functions declared as event handlers.

In a multi-threaded application using a real-time operating system (RTOS), only the time spent in the objects compiled with instrumentation is measured. Time spent in the scheduler/kernel and interrupt handlers is not reported. In the HTML-formatted report produced by the Reporter Tool, the *percentage of time* field is a percentage of the profiled time, not the absolute time that the application was running.

Behavior that Interferes With Instrumented Profiling

Several features of the C and C++ programming languages can have an impact on profiling results. The following features can result in unexpected results from profiling:

- *Unexpected termination of application.* If the application terminates unexpectedly, a complete set of profiling information may not be available. To ensure the profiling information is complete, all threads of execution should terminate by unwinding their stack (returning from `main()` or their thread creation function), or by calling `exit()`. RTOS-based systems may use a different implementation of `exit()`, so may require that data be flushed explicitly.

- *Unexpected flow control.* Functions that perform unexpected flow control, such as interrupts, C `setjmp/longjmp`, C++ exceptions, or calling other instrumented functions via `asm()` statements, may result in inaccurate profiling information. Instrumented profiling relies on the typical C/C++ behavior of call/return to be able to measure cycle counts in functions. When features, such as `setjmp` or C++ exceptions, return through multiple stack frames, instrumented profiling attempts to complete the profiling information for any stack frames unwound, but this may be inaccurate.

Profile-Guided Optimization and Code Coverage

The data recorded when running an application built with profile-guided optimization (see Using Profile-Guided Optimization) can also be used to generate a code coverage report using the IDE's Reporter Tool. A code coverage report provides a listing of your application's C/C++ source with execution counts for individual lines of code. To produce a code coverage report:

1. Compile the application for profile-guided optimization for either simulators (see Using Profile-Guided Optimization With a Simulator) or hardware (see Using Profile-Guided Optimization With Hardware).

2. Run the application to produce a `.pgo` file.

3. Select *File > New > Code Analysis Report*.

4. Ensure that *Code coverage* is selected.

5. Enter the name of the application executable in the ***DXE that produced the data*** field.

6. If the application `.ldf` file does not contain an `EXECUTABLE_NAME` directive, the ***Data file*** field will not have been automatically updated. Enter the name of the `.pgo` profiling data file into the field.

7. Enter the filename for the HTML report that will be generated.

8. Click *Finish*.

To generate a report from the command line, in either pre-formatted HTML or unformatted XML, refer to the Reporter.exe Command Line Report Generation Utility.

Code Coverage Report

The code coverage report contains a function-by-function summary of the application. For each C and C++ source file compiled with profile-guided optimization, a line count will be displayed, indicating how many times that line was executed.

Unexpected Line Counts in a Code Coverage Report

Several compiler features may impact the accuracy of a code coverage report. Compiler optimizations may rearrange code for better efficiency, and in some cases remove sections of code. This may result in unexpected line count information being displayed in the code coverage report.

If the application was compiled for profile-guided optimization on hardware, no line count information will be reported for any function declared with an interrupt handler pragma.

If the `.pgo` file already exists when you run your application to gather a profile, the new profile data will accumulate into the same existing `.pgo` file rather than replacing it. This allows you to run your application under a number of different conditions and gather an overall coverage report.

Heap Debugging

The support for heaps provides convenient access to dynamic memory within an application. While this is an easy and efficient way to use dynamic memory, the lack of bounds checking associated with pointer accesses means that mistakes are easy to make, and may have unpredictable side effects which can be hard to identify and debug. CCES provides a heap debugging library which can be used to detect errors in the use of the heap, helping identify issues which may be causing unintended behavior.

The heap debugging library constrains debug versions of the heap manipulation functions (such as `malloc`, `free`, `new`, `delete`) provided by the C and C++ run-time libraries, which record the heap activity and attempt to identify any potential issues with the usage of the heap, such as writing beyond the bounds of a buffer or failing to free memory.

The heap debugging library maintains a record of allocated blocks within the heap to track the current state of the heap. This recorded information is used as a reference to ensure that any heap allocations are valid; for example, checking that the block that is being freed has been allocated by `calloc`, `malloc`, `realloc`, `new` or any derivatives and has not been freed previously. A guard region of 12 bytes, filled with a known bit pattern, is written before and after each block allocated from the heap and is checked at deallocation to detect any overwriting of the bounds of the block. These bit patterns can be changed at build-time or runtime to avoid the bit patterns corresponding to any application data that may be written into them, causing the bounds overflow to go undetected.

A cleanup function, adi_heap_debug_end, detects any potential memory leaks (memory that has been allocated but not deallocated) and heap corruption. This function is registered via `atexit`, and is invoked if an application calls `exit` or returns from `main`.

The heap debugging library has the ability to generate a report detailing heap usage and errors via the Reporter Tool, to provide diagnostics via `stderr` at runtime, to check heap(s) for corruption, and to generate a current heap state snapshot of the heap(s).

The heap debugging library can be used simply by being linked in with your application, meaning that source code does not need to be re-built. The heap debugging library also contains additional functions to allow the behavior of the heap debugging to be modified or for additional diagnostic tests to be carried out at runtime. These additional functions can be used by including the header, heap_debug.h, and require your code to be re-built. See the C/C++ Run-Time Library chapter for more information.

The heap debugging library can enabled in the IDE by selecting *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Run-time Checks > Link against heap debugging libraries*.

For a comprehensive list of errors detected by the heap debugging library, refer to Detected Errors.

The heap debugging library will require additional memory for code and data, so an application may fail to link for projects which do not have sufficient additional memory available. Heap and stack usage is also increased so runtime errors may occur if insufficient stack or heap is available within your application.

The heap debugging library requires an underlying I/O device driver to produce output to either `stderr` or the `.hpl` file, as described in Device for Profiling Output.

Calls to heap allocation and deallocation functions will also take longer when heap debugging is enabled than if it is disabled, especially if report generation is enabled.

- Getting Started With Heap Debugging

- Using Heap Debugging Library

- Generating a Heap Debugging Report

Getting Started With Heap Debugging

To use heap debugging, you first need to link your application against the heap debugging library instead of the normal heap library. You may also need to modify your application to perform some initial configuration, depending on whether your application is single- or multi-threaded, and levels of logging and diagnostics you require. This section contains:

- Linking With the Heap Debugging Library, which covers how to activate the heap debugging library

- Heap Debugging Macro, which explains how you can conditionally include configuration code in your application

- Default Behavior, which describes how the out-of-the-box configurations for the heap debugging library

- Additional Heap Overheads, which gives a brief summary of the extra data requirements of heap debugging

- Heap Debugging Report, which identifies the file produced by the heap debugging library

Linking With the Heap Debugging Library

You can enable the heap debugging library:

- In the IDE by selecting *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Run-time Checks > Link against heap debugging libraries*.

- On the command-line, via the -rtcheck-heap switch.

Heap Debugging Macro

When *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Run-time Checks > Link against heap debugging libraries* has been selected, the macro `_HEAP_DEBUG` is defined in the compiler, assembler, and linker.

This macro is used in the header file heap_debug.h to define either prototype functions when enabled, or to use macros to replace any heap debugging-specific function calls with statements mimicking a successful return from that function. This allows code to work independently of the heap debugging library being linked, with minimal performance overhead when the heap debugging library is not used.

The `_HEAP_DEBUG` macro is also used to control the linking of the heap debugging library in the default `.ldf` files.

Default Behavior

The behavior of the heap debugging libraries can be configured either at build-time or at runtime. The *Default Configuration for Heap Debugging* table shows the default configuration.

Table 3-15: Default Configuration for Heap Debugging

Generate `.hpl` log file	Enabled
Generate diagnostics to `stderr`	Disabled

The choice of configuration will affect the run-time performance of the application. For example, an application configured to log all heap activity to a file will make far more calls to the I/O library than an application configured only to emit an error diagnostic when a problem is encountered. However, the choice of configuration does not affect the additional code/data requirements imposed, as the heap debugging library has to record the same information in order to detect errors, regardless of whether that information is also being written to an activity log.

By default, applications generate an .hpl file of the heap activity; see Heap Debugging Report. The file can be converted into an HTML report for later analysis.

By default, no diagnostics regarding heap usage will be written to stderr. You can enable stderr diagnostics by calling:

```
adi_heap_debug_enable(_HEAP_STDERR_DIAG);
```

If your application does not terminate via exit or by returning from main, the heap debugging cannot track memory leaks or some cases of heap corruption. You will need to call adi_heap_debug_end at a suitable point in the application. Calling adi_heap_debug_end will instruct the heap debugging library to check for any memory leaks and corruption before cleaning up any internal data used.

If adi_heap_debug_end is not called either manually or via exit, memory leaks can be identified in the report by the presence of a memory allocation without a corresponding deallocation. Heap corruption can be detected by calling adi_verify_all_heaps from anywhere within your application.

Additional Heap Overheads

The over-allocation of each memory block by 24 bytes is used as a guard region around the block. In addition, the heap debugging library uses the system heap to allocate memory used for internal data. Approximately 24 bytes of memory are allocated from the system heap, per allocation made from any heap. And 24 bytes of memory are allocated from the system heap to record information about each heap in the system.

Heap Debugging Report

The heap debugging library uses the symbol __executable_name, provided by the EXECUTABLE_NAME() LDF directive to determine the name of the .hpl file used to generate the heap debugging report. If the __executable_name symbol is not present, the file unknown.hpl will be used. For more information, see Application Analysis and File Naming.

Using Heap Debugging Library

The following sections describe the use of the heap debugging library. They detail the type of issues detected by the heap debugging library, explain how to view the library's diagnostics, and how refine the diagnostics according to your needs. Topics include:

- Detected Errors lists the issues that the heap debugging library can detect.

- Viewing Reports explains how to convert the generated .hpl log file into report in HTML format.

- stderr Diagnostics covers how to control diagnostics emitted to the standard error stream.

- Call Stack discusses the call stack recorded with each heap operation, and how to configure this.

- [Setting the Severity of Error Messages](#) explains how to change the severity of each encountered issue.

- [Default Diagnostic Severities](#) lists the severity levels used by default.

- [Guard Regions](#) discusses the memory spaces allocated before and after each heap block, to detect writes beyond the block boundaries.

- [Enabling and Disabling Features](#) explains how to configure the library at build-time and at runtime.

- [Buffering](#) covers setting up a buffer to capture heap information while I/O is not possible.

- [Pausing Heap Debugging](#) explains how the tracing may be temporarily suspended.

- [Finishing Heap Debugging](#) gives advice on ensuring that all your heap tracing information is flushed to the log file.

- [Verifying Heaps](#) describes how you can programmatically ensure that your heaps are consistent.

- [Behavior of Heap Debugging Library](#) notes that using the heap debugging library will have an impact on the characteristics of your application.

- [Unfreed File I/O Buffers](#) explains a side-effect of the inter-dependence between the heap library and I/O library.

- [Memory Used by Operating Systems](#) indicates how any heap usage by the RTOS may lead to additional entries in the heap log.

Detected Errors

The following errors will be detected by the heap debugging library:

- Allocation of length zero

- Allocations that are bigger than the heap

- deallocation of a previously de-allocated memory

- deallocation of a pointer not returned by an allocation function

- `delete[]` of memory allocated by `new`

- `delete[]` of memory allocated by C functions (`calloc`, `malloc`, `realloc`)

- `delete` of memory allocated by C functions (`calloc`, `malloc`, `realloc`))

- `delete` of memory allocated by `new[]`

- `free` of memory allocated by C++ allocator operations

- `free` of null pointer

- `free` from incorrect heap

- Memory leaks (memory that has not been de-allocated)

- `realloc` of memory allocated by C++ allocator operations

- `realloc` of pointer not returned by allocation function

- `realloc` from incorrect heap

- Using heap functions from within an interrupt

- Writing beyond the scope of allocated memory block (up to 12 bytes before and after allocated memory)

- Writing to memory that has been de-allocated

Using the known bit patterns written in and around blocks by the heap debugging library can help to identify erroneous reads by the presence of these bit patterns in live data. These erroneous reads may be from:

- Memory that has been allocated from the heap but is uninitialized

- Memory that has been de-allocated

- Memory that is beyond the scope of the allocation (up to 12 bytes before or after allocated memory)

See Guard Regions for more information on these bit patterns.

Viewing Reports

To create an HTML report for your application's heap activity:

1. Build the application with *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Run-time Checks > Link against heap debugging libraries* or with -rtcheck-heap.

2. Run the application to produce a `.hpl` file.

3. Select *>File > New >Code Analysis Report*.

4. Ensure that *Heap debugging* is selected.

5. Enter the name of the application executable in the *DXE that produced the data* field.

6. If the application `.ldf` file does not contain an `EXECUTABLE _NAME` directive, the *Data file* field will not have been automatically updated. Enter the name of the `.hpl` profiling data file into the field.

7. Enter the filename for the HTML report that will be generated.

8. Click *>Finish*.

stderr Diagnostics

The heap debugging library can provide console diagnostic reporting for any issues detected with the heap usage, writing diagnostic messages to `stderr` as they are detected.

To enable `stderr` diagnostic reporting at runtime, call:

```
#include <heap_debug.h>
adi_heap_debug_enable(_HEAP_STDERR_DIAG);
```

To enable `stderr` diagnostics at build-time, define the following C variable in your application source:

```
bool adi_heap_debug_stderr_diag = true;
```

stderr diagnostics can have one of three severities: error, warning, or ignored.

Errors will print a diagnostic message and then call adi_fatal_error. For more information, see Fatal Error Handling in the C/C++ Run-Time Library chapter.

Warnings will print a diagnostic message and then continue the application as normal. Ignored errors will not produce any diagnostic messages and will not terminate the application.

The severity of errors will not have any impact on the content of the generated heap debugging output file (.hpl) or the heap debugging report generated from it; all errors is included.

Generated diagnostics will be in the form:

```
Heap [severity] in block [address]: [message]
```

when a memory address is relevant, or in the form:

```
Heap [severity]: [message]
```

when no memory address is relevant. Both will be followed by a call stack where one is known and relevant. severity will be either "error" or "warning", address represents the address of the memory block concerned, as returned by the allocation function. message will be a short description of the issue which has been detected.

The call stack reported will be the call stack of the function that identified the issue. This may not be the same function as the source of the error in some cases, such as detecting heap corruption and memory leaks.

Some examples of diagnostic messages are shown below.

Warning About Attempting to Free a Block That is Already Free

```
Heap warning in block 0xFF80647C: free of free block
Call stack: 0xFFA098AC 0xFFA080F6
```

Indicating that the memory block at address 0xFF80647C has been deallocated twice.

Warning About Calling malloc With Zero Size

```
Heap warning: allocation of length 0
Call stack: 0xFFA09972 0xFFA080F6
```

No block address has been provided as there is no address associated with this issue.

Warning About Memory Leak

```
Heap warning in block 0xFF80647C: unfreed block
```

No call stack is displayed here as it would refer to the call stack of the function in which the leak was detected.

Call Stack

The call stack associated with heap operations is included in the heap debugging output file (and the report generated from the heap debugging output file using the Reporter Tool) or any diagnostic messages produced by the heap debugging library in order to help identify the source of the identified issue.

The call stack is stored in a buffer on the system heap, requiring eight bytes of memory for each potential element in the call stack. By default, this call stack is five elements deep. The depth of the call stack can be changed by calling adi_heap_debug_set_call_stack_depth at runtime, which will try to re-allocate sufficient space for this buffer, or keeping the original buffer and returning false if it is not possible to change the call stack depth.

The values displayed in the call stack are the PC address of the return from the previous function, starting from the call to the heap function and traversing the stack towards `main` (up to the maximum call stack depth).

For example, when run, the following code

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

void do_free(char *x) {
    free(x);
}

void main(void) {
    adi_heap_debug_enable(_HEAP_STDERR_DIAG);
    do_free(0x0);
}
```

will produce the following warning:

```
Heap warning: free of null pointer
all stack: 0xFFA0153E 0xFFA01554 0xFFA00130
```

where the addresses in the call stack, `0xFFA0153E` and `0xFFA01554`, refer to the return from the call to `free` in `do_free` and the call to `do_free` from `main`, respectively. The last address, 0xFFA00130, is the call to main from the start-up code.

Setting the Severity of Error Messages

When `stderr` diagnostics are enabled, the severity of errors can be set, based on the type of the error. These severities are described in the *Heap Debugging Diagnostic Message Severities* table.

Table 3-16: Heap Debugging Diagnostic Message Severities

Severity	Description
Error	The application will print a diagnostic message and terminate
Ignored	The application will not print any diagnostic message and will continue running
Warning	The application will print a diagnostic message and continue running

These can be configured at runtime by calling the functions `adi_heap_debug_set_error`, `adi_heap_debug_set_ignore`, and `adi_heap_debug_set_warning`, with a parameter, which is a bit-field where each bit represents an error type. Macros representing these bits are provided by heap_debug.h. Multiple error types can be set to a severity at once by using the bitwise OR operator.

These severities can also be configured at build-time by defining the following bit fields using the macros defined in `heap_debug.h`:

```
unsigned long adi_heap_debug_error;
unsigned long adi_heap_debug_ignore;
unsigned long adi_heap_debug_warning;
```

> **NOTE:** Each error class should only be added to a single status, and each error class should be added to a status, otherwise unexpected behavior may occur.

These priorities have no impact on the report generation; all detected errors will still be displayed in the generated report.

If a warning is encountered, but the heap debugging library is unable to use I/O due to being in an interrupt or scheduling being disabled, the warning will be raised to an error and `adi_fatal_error` will be called. For this reason, setting the error type `_HEAP_ERROR_ISR` (heap usage within an ISR) to a warning will have no effect. Setting `_HEAP_ERROR_ISR` to be ignored will behave as expected.

Changing Error Severity Examples

To promote any cases in which the wrong heap is used to de-allocate memory (and any cases of attempting to allocate more memory than the size of the heap to be a terminating error), the following code can be used:

```
#include <heap_debug.h>
adi_heap_debug_set_error(_HEAP_ERROR_WRONG_HEAP |
                         _HEAP_ERROR_ALLOCATION_TOO_LARGE);
```

To demote any cases of using the wrong function to de-allocate memory, deallocations of invalid addresses and heap corruption to a warning, the following code can be used:

```
#include <heap_debug.h>
adi_heap_debug_set_warning(_HEAP_ERROR_FUNCTION_MISMATCH |
                           _HEAP_ERROR_INVALID_ADDRESS |
                           _HEAP_ERROR_BLOCK_IS_CORRUPT);
```

To ignore any cases of the wrong heap or wrong function being used to de-allocate memory, the following code can be used:

```
#include <heap_debug.h>
adi_heap_debug_set_ignore(_HEAP_ERROR_WRONG_HEAP |
                          _HEAP_ERROR_FUNCTION_MISMATCH);
```

Default Diagnostic Severities

By default, any potentially suspicious heap behavior which is documented as acceptable by the run-time libraries or C standard will result in a warning at runtime, since although this behavior may be intentional it may indicate an error in the usage of the heap, such as attempting to free memory from the wrong heap. Behavior which is incorrect will result in an error at runtime. No issues are ignored by default.

The default severities of error messages are detailed in the *Default Heap Debugging Diagnostic Severities* table.

Table 3-17: Default Heap Debugging Diagnostic Severities

Error Type	Default Severity
_HEAP_ERROR_UNKNOWN	Error
_HEAP_ERROR_FAILED	Warning
_HEAP_ERROR_ALLOCATION_OF_ZERO	Warning
_HEAP_ERROR_NULL_PTR	Warning
_HEAP_ERROR_INVALID_ADDRESS	Error
_HEAP_ERROR_BLOCK_IS_CORRUPT	Error
_HEAP_ERROR_FREE_OF_FREE	Warning
_HEAP_ERROR_FUNCTION_MISMATCH	Error
_HEAP_ERROR_UNFREED_BLOCK	Warning
_HEAP_ERROR_WRONG_HEAP	Warning
_HEAP_ERROR_ALLOCATION_TOO_LARGE	Warning
_HEAP_ERROR_INVALID_INPUT	Error
_HEAP_ERROR_INTERNAL	Error
_HEAP_ERROR_IN_ISR	Error
_HEAP_ERROR_MISSING_OUTPUT	Warning
_HEAP_ERROR_ADDRESSING_MISMATCH	Warning

For more information on error classes, see heap_debug.h.

Guard Regions

The heap debugging library uses guard regions of 12 bytes before and after each block allocated from the heap containing a known bit pattern. These patterns are checked when the free/allocated status of the block is modified or at the end of the application. If the values do not match, then heap corruption must have occurred, such as overwriting of a buffer or writing to a block which has been deallocated.

Blocks that have been allocated from the heap have 12 bytes before and after the block filled with the allocated block boundary pattern. Corruption of these before and after guard regions indicates underflow and overflow of the block, respectively.

The contents of allocated blocks (other than blocks allocated using `calloc`) are filled with the allocated block contents pattern to help manually identify the use of allocated but uninitialized memory.

Free blocks are filled with the free blocks pattern. The 12-byte guard region following the block is also filled with this value, though the 12-byte guard region before the block is not as these 12 bytes are used by the heap for the operation of the free list. Corruption of this memory indicates that memory has been written to after it has been deallocated.

Reading beyond the scope of the allocated block, free or uninitialized memory can be identified by these bit patterns appearing in live data within the application.

The default bit-patterns for the guard regions are shown in the *Heap Debugging Guard Region Values* table.

Table 3-18: Heap Debugging Guard Region Values

Guard Region	*Bit Pattern*
Free blocks	`0xBD`
Allocated block boundaries	`0xDD`
Allocated block contents (not `calloc`)	`0xED`
Allocated block contents (`calloc`)	`0x00`

These patterns can be changed at runtime by calling:

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

where each parameter is a character representing the required bit pattern. Any existing blocks will be checked for corruption before the pattern is changed. If there are any corruptions, then `adi_heap_debug_set_guard_region` does not change the guard regions and returns false. If the heap is valid, then the guard regions for all existing allocations will be changed along with the guard regions of any future allocations. The patterns written to allocated block contents will not be updated, though any new allocations will be filled with the new bit pattern.

The patterns can also be overridden at build-time by defining the appropriate "C" variable, shown in the *Heap Debugging Guard Region Variables* table.

Table 3-19: Heap Debugging Guard Region Variables

Guard Region	*Variable*
Free blocks	`unsigned char adi_heap_guard_free`
Allocated block boundaries	`unsigned char adi_heap_guard_alloc`
Allocated block contents (not `calloc`)	`unsigned char adi_heap_guard_content`

These variables will be updated if adi_heap_debug_set_guard_region is called at runtime, so they can be used to identify the current guard region values.

   NOTE: The variables described in the *Heap Debugging Guard Region Values* table should not be written to at runtime, or false corruption errors may be reported.

The guard regions can be returned to the defaults detailed in the *Heap Debugging Guard Region Values* table by calling `adi_heap_debug_set_guard_region`. As with `adi_heap_debug_set_guard_region`, `adi_heap_debug_reset_guard_region` only changes the guard regions if no corruption is detected.

Enabling and Disabling Features

There are two ways in which features can be configured within an application: via function calls at runtime, or by defining variables at build time. The default configuration is described in Default Behavior.

The features that can be enabled or disabled, along with the macros, provided by `heap_debug.h`, are detailed in the *Configurable Heap Debugging Features* table.

NOTE: Any allocation or deallocation made while heap debugging is disabled is not recorded by the heap debugging library. This may result in errors if the memory is then manipulated with heap debugging enabled. For instance, a block allocated with heap debugging disabled and then de-allocated when heap debugging has been enabled will report a free from invalid address error. Conversely, allocation of blocks with heap debugging enabled and then manipulation of those blocks with heap debugging disabled may result in an unfreed block error.

Table 3-20: Configurable Heap Debugging Features

Feature	*Macro*
Run-time diagnostics	`_HEAP_STDERR_DIAG`
Generation of .hpl file for heap report	`_HEAP_HPL_GEN`
Tracking of heap usage	`_HEAP_TRACK_USAGE`

At Runtime

Features within the heap debugging library can be enabled or disabled at runtime by using:

- The adi_heap_debug_enable or adi_heap_debug_disable function, respectively

- A bitfield, constructed by combining the required macros specified in the *Configurable Heap Debugging Features* table (see Enabling and Disabling Features)

- The bitwise `OR` operator. To enable both run-time diagnostics and `.hpl` file generation, the following can be used:

```
adi_heap_debug_enable(_HEAP_STDERR_DIAG | _HEAP_HPL_GEN);
```

Enabling either runtime diagnostics or `.hpl` file generation implicitly enables tracking of heap usage.

At Build Time

The global variables used to configure the heap debugging features can be defined at build time, allowing the default configuration to be modified with no performance overheads. These values can also be read at runtime to identify the current configuration. These variables are detailed in the *Variables Used to Configure Heap Debugging Features* table.

NOTE: Do not write to the variables directly at runtime, or unexpected behavior can result.

Table 3-21: Variables Used to Configure Heap Debugging Features

Feature	Variable
Tracking of heap usage	`bool adi_heap_debug_enabled`
Run-time diagnostics	`bool adi_heap_debug_errors_enabled`
Generation of `.hpl` file for heap report	`bool adi_heap_debug_hpl_gen`

Buffering

The contents of the `.hpl` file used to generate a heap debugging report can be buffered by the heap debugging library to improve performance and to avoid any recorded data being lost when it is not currently safe to write to that file.

The buffer will be flushed periodically or when it is safe to carry out I/O and the buffer has reached a certain threshold.

By default, the heap debugging library does not have a buffer configured. This means that every use of the heap will result in the data being written to the output file. As a result, the output file is always up to date and no flushing of the output is required. This does, however, have an impact on execution time due to the overhead of the I/O operations required and means that any data that cannot be written at the time will be lost.

A buffer can be specified at runtime by calling adi_heap_debug_set_buffer with a pointer to the memory and the size of the buffer in addressable units. The buffer threshold will be set to half of the size of the buffer.

A buffer can be configured at build-time by defining the variables described in the *Variables Used to Configure Heap Debugging Buffer* table.

The macro _ADI_HEAP_MIN_BUFFER, provided by `heap_debug.h`, can be used to determine the minimum size required for the heap debugging output buffer to be usable. This macro represents the size required to store two entries of the log data along with associated call stacks. The memory requirement for an entry of log data is 56 bytes + 8 bytes per call stack item, up to the maximum call stack depth. The default maximum call stack depth is five and can be modified by using `adi_heap_debug_set_call_stack_depth`.

When heap debugging is not enabled, _ADI_HEAP_MIN_BUFFER will be defined to 0.

Table 3-22: Variables Used to Configure Heap Debugging Buffer

Variable	Description
`void *adi_hpl_buf_ptr`	Pointer to the start of the buffer
`int adi_hpl_buf_size`	Size of the buffer in addressable units
`int adi_hpl_buf_threshold`	Threshold at which buffer will be flushed

The number of bytes of data that has been lost due to insufficient buffering is stored in the 32-bit integer variable `adi_hpl_buf_lost_data`, provided by `heap_debug.h`.

Pausing Heap Debugging

Heap debugging can be temporarily disabled at runtime to improve the performance in sections of code where heap usage does not need to be debugged. With debugging disabled, no checks will be carried out and no allocations or deallocations will be recorded, but performance will be close to the non-debug version of the heap functions.

> **NOTE:** Heap debugging is enabled and disabled globally, so pausing heap debugging will affect the tracking of all heap usage across any threads which are running until it has been re-enabled.

> **NOTE:** Any allocations or deallocations made while heap debugging was paused will not be recorded, so any corresponding operations made after heap debugging has been resumed may result in false errors being produced regarding invalid addresses or memory leaks.

Heap debugging can be paused by calling `adi_heap_debug_pause` and can be re-enabled by calling `adi_heap_debug_resume`.

Finishing Heap Debugging

If an application does not exit, or uses an operating system that does not support `atexit`, the heap debugging library will not be able to clean up or check for corrupt blocks and memory leaks. In these cases, the clean up function `adi_heap_debug_end` should be called at a suitable point within your application. Heap debugging will be disabled upon completion of this function, and any further heap usage will be ignored unless heap debugging has been re-enabled by calling `adi_heap_debug_enable`.

It is safe to call `adi_heap_debug_end` multiple times within an application. If a .hpl output file has already been written to by the current instance of the application, then the output file will be appended to.

> **NOTE:** `adi_heap_debug_end` will attempt to flush any buffer for the `.hpl` file generation, so it should only be called when I/O is safe to use. Calling `adi_heap_debug_end` from within an interrupt or unscheduled region will result in `adi_fatal_error` being called.

Verifying Heaps

It is possible to check that a heap or that all heaps are free of corruption (see Guard Regions for more information on heap corruption) at runtime by calling the functions adi_verify_heap or adi_verify_all_heaps, respectively.

These functions return true if the heap or heaps are free of corruption, or false if corruption is detected.

Behavior of Heap Debugging Library

While the heap debugging library is compatible with the non-debug functionality where possible, so that application should operate in the same with heap debugging enabled as without, some minor changes in behavior may be observed. These changes in behavior are detailed below.

Application Size

Due to the additional functionality provided by the heap debugging library, code and data usage for your application increase when using the heap debugging libraries. If insufficient space is available for this library, your application can fail to link.

Performance

Due to extra validation checks, performance in the heap manipulations will be degraded compared to the non-debug version of the functions provided by the C/C++ run-time libraries, especially if generation of the `.hpl` file is enabled. With heap debugging disabled or paused, the performance should be close to the non-debug version of the heap manipulation functions.

Heap debugging can be enabled or disabled at run-time, allowing you to ignore selected parts of your applications to minimize the impact of heap performance overheads.

NOTE: Heap operations, which are carried out when heap debugging is disabled, are ignored and can result in false errors being reported.

By default, for non-threaded applications, an output file is created which is used to generate a heap debugging report. The I/O operations required for this are time-consuming and can be disabled to improve performance by using the following:

```
adi_heap_debug_disable(_HEAP_HPL_GEN);
```

If heap tracing is disabled, then run-time diagnostics should be enabled in order to identify any heap errors.

Heap Usage

For each allocation on any heap, the heap debugging libraries will over-allocate the memory by 24 bytes for use as a guard region, as well as approximately 24 bytes of internal data on the system heap. As a result, more heap space will be used when heap debugging is enabled. You may need to increase the size of your heaps if insufficient space is available.

Stack Usage

The additional function calls used for the heap debugging will make use of the stack of parameters and local variables, so the overall stack usage in your application will increase when using the heap debugging library, particularly when writing diagnostics or the `.hpl` file.

realloc

The versions of `realloc` and `heap_realloc` provided by the heap debugging library will always de-allocate the original block of memory and allocate a new block of memory of the required size; the equivalent of calling `malloc`, `free` then `memcpy`, while the non-debug versions of `realloc` and `heap_realloc` will try to re-use the existing memory first.

This change in behavior with the heap debugging version is to catch cases where a block has been reallocated but pointers have not been updated to reference the new block. These cases may happen to work in an application, but this behavior cannot be relied on and may result in unexpected behavior.

As a result of this, some calls to `realloc` or `heap_realloc` may fail with the heap debugging which are successful without. This can be avoided by ensuring sufficient heap space is available.

Unfreed File I/O Buffers

For each file stream used, the run-time library allocates 512 bytes of memory from the heap to use as a buffer. For reasons of performance and code size, the run-time libraries do not free this memory upon application exit. The

heap debugging library will identify these blocks as belonging to a file buffer so it will not report an error about being unfreed. The allocation of the I/O buffer memory will be seen in the heap debugging report without a corresponding free.

Memory Used by Operating Systems

Operating systems used in an application may use the heaps to store internal data. This data may be reported as an unfreed block by the heap debugging library as it cannot identify the source of the allocation. Some unfreed block reports are to be expected when using an operating system if it is still running.

Generating a Heap Debugging Report

The default output format for the Heap Debugging report is HTML. To produce the HTML file:

1. Choose *File > New > Code Analysis Report*.

   The *Analysis Report* wizard appears.

2. Select *Heap Debugging* and click *Next*.

   The *Input/output files for code analysis report generation* page appears.

3. In *DXE that produced the data file*, enter the name of the application executable file. Alternately, browse to the file.

4. In *Data file*, enter the name of the heap debugging (`.hpl`) data file. Alternately, browse to the file.

   ADDITIONAL INFORMATION: If the application `.ldf` file does not contain an `EXECUTABLE_NAME` directive, the *Data file* filed is not populated automatically.

5. In *Report document's output file name*, enter the name of the new HTML report file. Alternately, browse to the existing file.

6. Click *Finish*.

To generate a report from the command line, in either pre-formatted HTML or unformatted XML, refer to the Reporter.exe Command Line Report Generation Utility.

Stack Overflow Detection

The compiler provides support for detecting stack overflows, which can be particularly troublesome bugs in the limited environment of an embedded system.

This section includes:

• About Stack Overflows gives a description of a stack overflow

• Compiler's Stack Overflow Detection Facility explains how to use the compiler's support for detecting stack overflows

About Stack Overflows

This section gives an introduction to stack overflows, and why they are problematic.

This section includes:

- What is Stack Overflow? describes a stack overflow, and why it is different from other bugs.

- Likely Causes of Stack Overflow gives examples of the issues that can lead to stack overflows.

- Difficulties in Diagnosing Stack Overflow shows why compiler support is useful.

- Limitations on the Compiler's Stack Detection Capability notes when compiler support is less useful.

- Fixing a Stack Overflow gives advice on what to do when you have located your stack overflow.

What is Stack Overflow?

A stack overflow is caused by the stack not being large enough for the application. The effects of a stack overflow are undefined; the effects can vary from data corruption to a catastrophic software crash.

The stack overflows when the stack pointer (SP) is modified to point past the end of the memory reserved for the stack and the stack is written to using the stack pointer or frame pointer (FP).

> **NOTE:** A stack overflow is different from stack corruption caused by a bug in your program code.

When the stack overflows, any writes to the stack using the stack pointer (SP) or the frame pointer (FP) will begin to corrupt an area of memory which it should not. The results are undefined.

Likely Causes of Stack Overflow

There are many reasons why a stack overflow can occur, For example,

1. A function defines a too-large local array.

2. A function defines a too-large variable-length array. Refer to Variable-Length Arrays in the Compiler chapter for more information.

3. A function uses the `alloca()` function, with an too-large value as its parameter, to allocate space in the stack frame of the caller. (Refer to Compiler Built-In Functions in the Compiler chapter.)

4. The `.ldf` file has insufficient space set aside for the stack.

5. A function calls itself recursively too many times.

6. A function's call tree is too deep.

7. A re-entrant interrupt handler is called too many times before the interrupt is fully serviced.

Note that the application only has to exceed its available stack space by one location for corruption to occur, so there may not be much difference between a safe stack usage and an overflow.

Difficulties in Diagnosing Stack Overflow

Without compiler support, debugging a stack overflow is not often easy and mostly involves setting breakpoints or adding tracing statements at various places in your application. A stack overflow might also not become apparent if

you are building your application in a Release configuration, when optimizations are enabled; a stack overflow might not reveal itself until your application is built in a Debug configuration, when optimizations are not enabled.

The timing of interrupts will also mask a stack overflow. If nested interrupts are enabled and the time taken to service the interrupts is insufficient before another interrupt is raised and serviced, then a stack overflow can occur.

Compiler's Stack Overflow Detection Facility

You can enable stack overflow detection via Project > Properties > C/C++ Build > *Settings > Tool Settings > Compiler > Run-time Checks > Generate code to catch a Stack Overflow*, or via the `-rtcheck-stack` -rtcheck-stack switch.

Once the compiler's stack overflow detection facility has been enabled, the compiler will generate code in the function's prologue and whenever the stack pointer (`SP`) is modified in the function code, to check that the stack pointer has not exceeded the stack limit. The current stack limit is held in a global data structure called `__adi_stack_bounds`.

If the stack pointer, once modified, exceeds the stack limit a function, called `_adi_stack_overflowed`, is invoked. The function that triggered the stack overflow can be discovered by examining the `RETS` register.

Limitations on the Compiler's Stack Detection Capability

The compiler cannot generate stack overflow detection code for assembly files or files that have already been compiled to object files.

For BF5xx and BF6xx parts the stack detection only checks for stack overflow due to register pressure. As such any stack underflows or allocations which cause the stack pointer to wrap around the address space will not be detected. This is not the case for BF7xx parts where both overflows and underflows will be detected.

Certain compiler features will cause the compiler to generate calls to support libraries, which will transiently use arbitrarily-deep call-trees, requiring additional stack space. These support libraries are not supplied with variants that include stack overflow detection, so these features should not be used in conjunction with stack overflow detection. These features are:

- Profiling With Instrumented Code

- Profile-Guided Optimization and Code Coverage

- Heap Debugging

Fixing a Stack Overflow

Once it has been identified that a stack overflow is the cause of your application failure, correcting the problem can be as simple as increasing the amount of memory reserved for your stack.

If, due to hardware memory restrictions, you are unable to increase the amount of memory used for the stack, then conduct a review of your application, examining your use of local arrays, alloa, function calling and other program code that leads to a stack overflow.

Uses of alloca in inlined functions may result in increased stack usage compared to the same functions being used out-of-line. If these are the cause of your stack overflow issues then you can avoid them by either preventing the

functions being inlined (via the never_inline pragma), or allocating the memory via another method such as the heap.

Reporter.exe Command Line Report Generation Utility

Reporter.exe is the command line utility used to generate reports for Instrumented Profiling, Code Coverage and Heap debugging. The default output format is a pre-formatted HTML report, but the un-formatted XML can be emitted using the -xmlswitch.

The following options are supported:

Switch	Usage
-dxe <path>	Path to the executable being profiled.
	This should not be re-built after generation of the report as the debug info may become out of date.
-printer:<printer-name> <data file>[, <data file>]	Specifies the report type to be generated and one or more input data files.
	For Code Coverage, use -printer:ReporterPGO. For Instrumented Profiling, use -printer:ReporterInstrProf. For Heap Debugging, use -printer:ReporterHeap.
-output <path>	Specifies the path of the generated report.
-xml	Generates an XML report. This is written to the base name of the output file with a .xml extension.
-help	Shows usage and command line information

For example, to produce an XML Code Coverage report, you would invoke the Reporter.exe tool as follows:

```
<path-to-install>/Reporter.exe -dxe Application.dxe -printer:ReporterPGO
Application.pgo -output report.xml -xml
```

4 C/C++ Run-Time Library

The C and C++ run-time libraries are collections of functions, macros, and class templates that may be called from your source programs. The libraries provide a broad range of services, including those that are basic to the languages such as memory allocation, character and string conversions, and math calculations. Using the library simplifies software development by providing code for a variety of common needs.

This chapter contains:

- C and C++ Run-Time Library 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 ccblkfn compiler.

- Documented Library Functions tabulates the functions that are defined by ANSI standard header files.

- C Run-Time Library Reference provides reference information about the C run-time library functions included with this release of the ccblkfn compiler.

The ccblkfn 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 full standard C++ library conforming to the ISO/IEC 14882:2003 C++ standard and the abridged C++ library, which is a conforming subset of the standard C++ library. The abridged C++ library includes the embedded C++ and standard template libraries.

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

NOTE: For more information on the C standard library, see *The Standard C Library* by P.J. Plauger, Prentice Hall, 1992. For 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.

NOTE: Selected libc functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

C and C++ Run-Time Library Guide

The C/C++ run-time libraries contain functions that can be called from your source. This section describes how to use the library and provides information on these topics:

- Calling Library Functions
- Using the Compiler's Built-In Functions
- Linking Library Functions
- Library Attributes
- Library Function Re-Entrancy and Thread Safety
- Working With Library Header Files
- Calling a Library Function From an ISR
- C++ Library Support
- File I/O Support
- Fatal Error Handling

For information about the C library's contents, see C Run-Time Library Reference.

For information about the ISO/IEC 14882:2003 C++ standard library and the abridged C++ library's contents, see C++ Library Support.

Calling Library Functions

To use a C/C++ library function, call the function by name and provide the appropriate arguments. The names and arguments for each function are described on the reference pages, which begin in C Run-Time Library Reference.

Like other functions, library functions should be declared. Declarations are supplied in header files, as described in 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.

- For C functions, this is an underscore (_) at the beginning of the C function name. For example, the C function main() is referred to as _main from an assembly program.

- Functions in C++ modules are normally compiled with an encoded function name. Function names in C++ contain abbreviations for the parameters to the function and also the return type. As such, they can become very large. The compiler "mangles" these names to a shorter form. You can instruct the C++ compiler to use the single-underscore convention from C, as shown by the following example.

```
extern "C" {
    int myfunc(int);    // external name is _myfunc
}
```

Alternatively, compile C++ files to assembler, and see how the function has been declared in the assembly file.

It may not be possible to call inline functions as the compiler may have removed the definition of the function if all calls to the function are inlined. Global static variables cannot be referred to in assembly routines as their names are encrypted.

For more information on naming conventions, see Symbol Names in C/C++ and Assembly in the Compiler chapter.

> **NOTE:** Create a CCES project or use the archiver (`elfar`), described in the *Linker and Utilities Manual*, to build library archive files of your own functions.

Using the Compiler's Built-In Functions

The C/C++ compiler's *built-in functions* are a set of functions that the compiler immediately recognizes and replaces with inline assembly code instead of a function call. Typically, inline assembly code is faster than a library routine, and does not incur the calling overhead. For example, the absolute value function, `abs()`, is recognized by the compiler, which subsequently replaces a call to the C/C++ run-time library version with an inline version.

To use built-in functions, include the appropriate headers in your source; otherwise, your program build will fail at link-time.

> **NOTE:** Standard math functions, such as `abs`, `min`, and `max`, are implemented using compiler built-in functions. They perform as described in C Run-Time Library Reference and DSP Run-Time Library Reference.

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 the default Linker Description File (`ADSP-<your_target>.ldf`).

If you are not using the default `.ldf` file, then either add the appropriate library/libraries to the `.ldf` file used for your project, or use the compiler's `-l` switch to specify the library to be added to the link line. For example, the switches `-lc -ldsp` add `libc.dlb` and `libdsp.dlb` to the list of libraries to be searched by the linker. For more information on the `.ldf` file, see the *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 provides a variant of `LibIO` (the I/O run-time support library):

- `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_ADI_FX_LIBIO`.

Library Location

The C/C++ run-time libraries are provided in binary form in directories named `Blackfin\lib\`*`process-`*
`or_rev_revision`:

- *processor* identifies the processor for which the library is built, and is the processor's name with the leading "ADSP-" stripped.

- *revision* identifies the silicon revision for which the library is built. For example, a revision of 0.1 would indicate that the library was built with the command-line switch `-si-revision 0.1`.

So the directory `Blackfin\lib\bf542_rev_any` contains libraries that have been built with `-proc ADSP-BF542 -si-revision any` switches.

The C/C++ run-time libraries are provided in source form, where available, in the directories named `Blackfin \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-BF542` would cause the compiler driver to read the `ADSP-BF542-compiler.xml` file in `System\ArchDef`.

Each such XML file indicates which library subdirectory should be used for each supported silicon revision of that processor. For example, the XML file for the ADSP-BF542 processor indicates that for silicon revision 0.2, the library directory to use is `Blackfin\lib\bf542_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 is `lib<name><attrs>.dlb`, where *name* indicates the library's purpose, and *attrs* is a sequence of zero or more attributes. The library's names are given in the *C/C++ Library Names* table, and the attributes are enumerated in the *Library Name Attributes* table.

Table 4-1: 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

Table 4-2: C/C++ Library Names

Description	Library Name	Comments
Compiler support library	libcc*.dlb	
C run-time libraries	libc*.dlb librt*.dlb libsmall*.lib	
C++ run-time library	libcpp*.dlb	
DSP run-time library	libdsp*.dlb	
Device driver/Services libraries	libdrv*.dlb libssl*.dlb libosal*.dlb	Refer to *System Services and Device Drivers* found in *System Run-Time Documentation*.
ETSI library	libetsi*.dlb	
Event library	libevent*.dlb	
Heap debugging library	libheapdbg*.dlb	
Instrumented profiling library	libprofile*.dlb	
I/O run-time library	libio*.dlb	
I/O run-time library with full support for the fixed-point format specifiers	libiofx*.dlb	

Table 4-2: C/C++ Library Names (Continued)

Description	Library Name	Comments
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.

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 crt<*procid*><*attrs*>.doj:

- *procid* indicates which processor the startup code is for; this is the last three 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 ***CRT File Name Suffices*** table.

Table 4-3: CRT File Name Suffices

crt File Name Suffix	Description
c	Startup file used for C++ applications
f	Startup file that enables file I/O support via stdio.h
s	Startup file used by applications that run in supervisor mode

Library Attributes

The run-time libraries make use of file attributes. (See File Attributes in the Compiler chapter for details on using file attributes.)

For each object (.obj) in the run-time libraries, the following is true:

Table 4-4: Run-Time Library Object Attributes

Attribute Name	Attribute Meaning and Value
libGroup	A potentially multi-valued attribute. Each value is the name of a header file that either defines obj or defines a function that calls obj.
libName	The name of the library that contains obj, without the processor and variant. For example, suppose that obj were part of libdsp.dlb, then the value of the attribute would be libdsp.

Table 4-4: Run-Time Library Object Attributes (Continued)

Attribute Name	Attribute Meaning and Value
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 I/O library functions fall into this category). 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. Default .ldf files use the prefersMem attribute in preference to the prefersMemNum attribute to specify the optimal 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

```
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` will have the attributes as defined in the *Attribute Values in libfunc.dlb* table.

Table 4-5: Attribute Values in libfunc.dlb

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

(1) `func1.doj` will not be performance-critical, based on its normal usage.

(2) `func2.doj` will be performance-critical in many applications, based on its normal usage.

(3) `libGroup` contains the union of the `libGroup` attributes of the two calling objects.

(4) `prefersMem` contains the highest priority of all the calling objects.

Exceptions to Library Attribute Conventions

This section lists exceptions to the library attribute conventions.

The C++ support libraries (`libcpp*.dlb` and `libcppfull*.dlb`) contain functions that have C++ linkage. C++ linkage implies that the entry point names within the libraries are encoded to include the parameter types, return type, and namespace within which the function is declared (this encoding is also known as *name mangling*). Thus, any C++ library function that is used as the value for a `libFunc` attribute must be the encoded name.

The *Additional libGroup Attributes* table lists additional `libGroup` attribute values.

Table 4-6: Additional libGroup Attributes

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
`runtime_checking`	Run-rime functions to provide support for dynamic checks
`stack_overflow_detection`	Run-rime functions to support detection of stack overflow
`libprofile`	Run-rime functions to support profiling

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

The *Default Memory Placement Summary* table summarizes the default memory placement using `prefersMem`.

Table 4-7: Default Memory Placement Summary

Library	*Placement*

`crt*.doj`
`crtn*.doj`
`cplbtab*.doj`
`mc_data*.doj`
	Hard placement using sections

Table 4-7: Default Memory Placement Summary (Continued)

Library	*Placement*

`libcpp*.dlb`
`libcppfull*.dlb`
`libetsi*.dlb`
	Any (`any`)

`libio*.dlb` `libprofile*.dlb`	External (`external`)
`libc*.dlb`	`any` except for the `stdio.h` functions that are `external` and `qsort`, which is `internal`
`libdsp*.dlb`	`internal` except for the windowing functions and functions that generate a twiddle table, which are `external`
`libevent*.dlb`	`internal` for anything that may be called in response to an event, plus `flush_data_buffer`; `external` for all exception idle loops (where the processor has to halt); `any` for functions that install and manage event handling functions
`libmc*.dlb`	Any (`any`)
`librt*.dlb`	`internal` or `any`
`libsmall*.dlb`	`any` or `external`, except for the vector table for processor events, which is `internal`

Mapping Objects to Flash Using Attributes

When using the memory initializer to initialize code and data areas from flash memory, be sure to map code and data (used during initialization to flash memory) so 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. Refer to the *Linker and Utilities Manual* for information on memory initialization.

Library Function Re-Entrancy and Thread Safety

This section includes the following topics:

- Non-Reentrant Functions
- Thread-Safe Libraries
- Using Thread-Safe Libraries

Non-Reentrant Functions

Many of the functions in the C/C++ run-time libraries are re-entrant, but some are not. A non-reentrant function can only have one active instance at any given time (that is, it has to return before it can safely be invoked again).

If multiple instances of a non-reentrant function are active at the same time, results are undefined. This can occur in the following situations:

- The function is invoked recursively, either directly or indirectly.
- An interrupt service routine (ISR) invokes a function while the main program or another ISR is also executing that function.

- Two or more threads in a multi-threaded program execute a function concurrently.

- Similarly to the previous case, more than one core in a multi-core program executes a function at the same time.

Non-reentrant functions are those which access global variables, including variables declared as `static`, or other global resources such as input/output devices.

Examples of such library functions include:

- `stdio.h` functions that operate on streams (but not those which operate on strings)

- C++ file streams

- Dynamic memory management functions, such as `malloc()` and `free()`

- `atexit()` and `signal()`, as they manipulate global handler tables

- Functions that return a result in a statically allocated buffer; for example, `time.h` functions such as `asctime()` or `localtime()`

- Functions that write to `errno`, such as many functions in `math.h`

- Functions that maintain private state across invocations; for example, `rand()` with its random seed and `strtok()` with a pointer to the last token

Invoking non-reentrant library functions from interrupt service routines is not supported. (It may be possible to do so safely in special circumstances; it is your responsibility to guarantee that only one invocation of the function is in progress at any given time.)

Thread-Safe Libraries

Use the thread-safe variants of the run-time libraries in multi-threaded programs. *Thread safety* means that functions that are non-reentrant can nevertheless be safely invoked from multiple threads.

This is achieved by two principal methods: thread-local storage and mutual exclusion. Where possible, thread-local storage is employed, whereby each thread gets its own version of global variables and buffers.

Where thread-local storage is not an appropriate solution, mutual exclusion is used to ensure that only one thread at a time can access shared global resources. This means that functions might block while waiting for another thread to release the resource in question. The following are affected:

- `stdio.h` streams and C++ file streams

- Dynamic memory management functions

- `atexit()` and `signal()`

The thread-safe variants of the run-time libraries have the suffix `"mt"` in their name. These are used both for multi-threaded and for multi-core programs.

Using Thread-Safe Libraries

Select the thread-safe libraries by specifying the `-threads` switch during compilation and linking. In the project *Properties* dialog box, this can be done by enabling the *Link against thread-safe libraries* option.

The effect of the `-threads` switch is to define the macro `_ADI_THREADS`. In the library headers, this macro selects some code that is specific to the thread-safe run-time libraries. Therefore, take care not to mix objects and libraries that have been compiled with and without the `-threads` switch. In the default Linker Description Files, the `_ADI_THREADS` macro selects the thread-safe variants of the run-time libraries.

The run-time library can be used in both single and multi-threaded environments. The thread-safe run-time libraries and other Analog Devices software use Analog Devices' own OS Abstraction Layer (OSAL). Each supported RTOS package includes its own implementation of the OSAL library which allows customers to use the run-time library seamlessly.

Working With Library Header Files

When using a library function in your program, include the function's header file with the `#include` preprocessor command. Each function's header file is identified in the *Synopsis* section of the function's reference page. Header files contain function prototypes, which are used by the compiler to check that the function is called with the correct arguments.

The *Standard C Run-Time Library Header Files* table shows the standard C run-time library header files supplied with this release of the Blackfin compiler. Refer to a C standard reference (see C/C++ Compiler Overview in the Compiler chapter) to augment information supplied in this chapter.

Table 4-8: Standard C Run-Time Library Header Files

Header	Purpose	Standard
adi_libldr.h	Dynamically-loadable modules	Analog extension
adi_types.h	Type definitions	Analog extension
assert.h	Diagnostics	ANSI
ccblkfn.h	Access to system facilities on Blackfin processors	Analog extension
ctype.h	Character handling	ANSI
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	Analog extension
iso646.h	Boolean operators	ANSI
libdyn.h	Deprecated support for dynamically-loadable modules	Analog extension
limits.h	Limits	ANSI
locale.h	Localization	ANSI

Table 4-8: Standard C Run-Time Library Header Files (Continued)

Header	Purpose	Standard
`math.h`	Mathematics	ANSI
`mc_data.h`	Routines for accessing the core-specific data for multi-core processors	Analog extension
`misra_types.h`	Exact-width integer types	MISRA-C:2004
`pgo_hw.h`	Profile-guided optimization support	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
`sys/adi_core.h`	Multi-core functions	Analog extension
`time.h`	Date and time	ANSI

The following sections describe 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`, and `float64_t`. The `adi_types.h` header file also includes `stdint.h` and `stdbool.h`.

assert.h

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

`filename:linenumberexpression` - Runtime Assertion

where:

- `filename` - name of the source file

- `linenumber` - current line number in the source file

- `expression` - expression tested

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

> **NOTE:** The strings associated with `assert.h` can be assigned to slower, more plentiful memory (thereby freeing up faster memory) by placing a `default_section` pragma above the sections of code that contain the asserts.
>
> For example,
> ```
> #pragma default_section(STRINGS,"sdram_bank1")
> ```
>
> This moves all strings - not just those associated with assert.
>
> Alternatively, place the `-section` flag on the compiler command line or include the option via *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Additional Options*.
>
> For example,
> ```
> -section strings=sdram_bank1
> ```

ccblkfn.h

The `ccblkfn.h` header file defines built-in functions that allow access to system facilities on Blackfin processors (see the *ccblkfn.h Library Functions* table in Documented Library Functions).

ctype.h

The `ctype.h` header file defines the functions that may be useful in classifying characters, or converting characters between upper and lower case.

All the functions defined by the header file have a single argument that is an `int` whose value is either `EOF` or a value that corresponds to an `unsigned char`. If the argument has some other value, then the behavior of the function will be undefined.

See the *Library Functions in time.h Header File* table in Documented Library Functions, which contains a list of the functions defined by this header file.

> **NOTE:** By default the `char` data type is signed and therefore the following may return an unexpected result:
> ```
> char ch = 0x80;
> int f = isdigit(ch);
>
> printf("isdigit(ch) = %d\n", f);
> ```

The scalar ch will be passed to isdigit as the value -128 (and not 128 as one may initially expect), which will lead to undefined behavior.

errno.h

The errno.h header file provides access to errno. 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 implemented by the compiler (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 available precision, 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 specified to be 32 bits wide, and they will have the same value as the macros for the long double data type when doubles are specified to be 64 bits wide. (See -double-size-{32 | 64}.)

heap_debug.h

The heap_debug.h header file defines a set of functions and macros for configuring and manipulating the heap debugging library. See Heap Debugging for more information on heap debugging.

When the macro _HEAP_DEBUG is not defined, the functions defined in heap_debug.h will be replaced by simple statements representing a successful return from that function so 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 ***Control Macros for Heap Debugging*** table.

Table 4-9: Control Macros for Heap Debugging

Macro	*Use*
_HEAP_STDERR_DIAG	Enable/disable diagnostics about heap usage via stderr
_HEAP_HPL_GEN	Enable/disable generation of .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_enable to enable or disable features at runtime. Tracking of heap usage will be implicitly enabled when either report generation or run-time

diagnostics are enabled at runtime. See Enabling and Disabling Features in the Optimal Performance from C/C++ Source Code chapter for more information.

Macros representing various categories of heap error are defined in `heap_debug.h`. These values can be used as parameters to the functions adi_heap_debug_set_error, adi_heap_debug_set_ignore, and adi_heap_debug_set_warning 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 run-time diagnostics are enabled. These error type macros are shown in the *Error Type Macros for Heap Debugging* table. See Setting the Severity of Error Messages in the Optimal Performance from C/C++ Source Code chapter for more information on using these macros.

Table 4-10: 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 does not 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 (that is, 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
`_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`	Report output has been lost due to insufficient or no buffer space
`_HEAP_ERROR_ADDRESSING_MISMATCH`	A block was deleted, freed or reallocated with a different address mode (-char-size-8/32) to where it was allocated.
`_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 the Optimal Performance from C/C++ Source Code chapter.

iso646.h

The `iso646.h` header file defines symbolic names for certain C (Boolean) operators. The *Symbolic Names Defined in iso646.h* table shows symbolic names and their associated value.

Table 4-11: 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. (See the -alttok switch in the *Compiler* chapter.)

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 adi_libldr.h APIs to load a DLM from a loader stream.

The `libdyn.h` header file contains type definitions and function declarations for loading dynamically-loadable modules (DLMs) that have been produced by the `elf2dyn` utility. For more information on using `elf2dyn`, refer to the *Loader and Utilities Manual*. For information on creating and using 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 a floating-point type.

locale.h

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

math.h

The `math.h` header file (see the ***Library Functions in the math.h Header File*** table in Documented Library Functions) includes power, trigonometric, 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`.

Some functions are also provided that support 16-bit and 32-bit fractional data types.

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

The -double-size-{32 | 64} compiler switch controls the size of the `double` data type. The default behavior is for the compiler to compile the `double` type as a 32-bit floating-point data type, and the header file will arrange that all references to a `double` function are directed to the equivalent `float` function (with the "f" suffix). Conversely, when the `double` type is defined as a 64-bit floating-point data type, all references to the `double` functions of this header file are directed to the `long double` version of the function (with the "d" suffix). This allows unsuffixed function names to be used with arguments of type `double`, regardless of whether `doubles` are 32 or 64 bits long.

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

Some functions in the `math.h` header file exist as both integer and floating point. The floating-point functions typically have an "f" prefix. Ensure that you are using the correct function.

NOTE: The C language provides implicit type conversion, so the following sequence produces surprising results with no warnings.

```
float x,y;
y = abs(x);
```

The value in x is truncated to an integer prior to calculating the absolute value; then it is reconverted to floating point for the assignment to y.

mc_data.h

The `mc_data.h` header file (see the ***Library Functions in the mc_data.h Header File*** table in Documented Library Functions) contains routines for accessing the core-specific data for multi-core processors.

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

When used with hardware (rather than a simulator), the `pgo_hw.h` header file declares user-callable functions in support of profile-guided optimization. For more information, see Profile-Guided Optimization and Code Coverage in the *Optimal Performance from C/C++ Source Code* chapter.

setjmp.h

The `setjmp.h` header file (see the ***Library Functions in the setjmp.h Header File*** table in Documented Library Functions) contains `setjmp` and `longjmp` for non-local jumps.

signal.h

The `signal.h` header file (see the ***Library Functions in the signal.h Header File*** table in Documented Library Functions) provides function prototypes for the standard ANSI `signal.h` routines.

stdarg.h

The `stdarg.h` header file (see the ***Library Functions in the stdarg.h Header File*** table in Documented Library Functions) contains definitions needed for functions that accept a variable number of arguments. Programs that call such functions must include a prototype for the referenced functions.

stdbool.h

The `stdbool.h` header file contains these Boolean-related macros (`true`, `false`, and `__bool_true_false_are_defined`) and an associated data type (`bool`). The `stdbool.h` header file is introduced in the C99 standard library.

stddef.h

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

stdfix.h

The `stdfix.h` file contains function prototypes and macro definitions to support the native fixed-point types `fract` and `accum` as defined by the ISO/IEC Technical Report 18037. The inclusion of this header file enables the `fract` and `accum` keywords as aliases for `_Fract` and `_Accum`, respectively. A discussion of support for native fixed-point types is given in Using Native Fixed-Point Types in the *Compiler* chapter.

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 is introduced in the C99 standard library.

The *Exact-Width Integer Types* table shows each of the typedefs defined by the header file, and documents the macro name of the associated minimum and maximum values for the types.

Table 4-12: Exact-Width Integer Types

Type	Common Equivalent	MIN	MAX
int8_t	signed char	INT8_MIN	INT8_MAX
int16_t	short	INT16_MIN	INT16_MAX
int32_t	int	INT32_MIN	INT32_MAX
int64_t	long long	INT64_MIN	INT64_MAX
uint8_t	unsigned char	0	UINT8_MAX
uint16_t	unsigned short	0	UINT16_MAX
uint32_t	unsigned int	0	UINT32_MAX
uint64_t	unsigned long long	0	UINT64_MAX
int_least8_t	signed char	INT_LEAST8_MIN	INT_LEAST8_MAX
int_least16_t	short	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 char	0	UINT_LEAST8_MAX
uint_least16_t	unsigned short	0	UNT_LEAST16_MAX
uint_least32_t	unsigned int	0	UNT_LEAST32_MAX
uint_least64_t	unsigned long long	0	UNT_LEAST64_MAX
int_fast8_t	signed char	INT_FAST8_MIN	INT_FAST8_MAX
int_fast16_t	short	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 char	0	UINT_FAST8_MAX
uint_fast16_t	unsigned short	0	UINT_FAST16_MAX
uint_fast32_t	unsigned int	0	UINT_FAST32_MAX
uint_fast64_t	unsigned long long	0	UINT_FAST64_MAX
intmax_t	long long	INTMAX_MIN	INTMAX_MAX
intptr_t	int	INTPTR_MIN	INTPTR_MAX
uintmax_t	unsigned long long	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 type-defs in other headings.

Table 4-13: 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()`, `INT16_C()`, `INT32_C()`, `UINT8_C()`, `UINT16_C()`, `UINT32_C()`, `INT64_C()`, and `UINT64_C()`.

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

stdio.h

The `stdio.h` header file (see the ***Supported Library Functions in the stdio.h Header File*** table in Documented Library Functions) defines a set of functions, macros, and data types for performing input and output. The library functions defined by this header file are thread-safe but they are not generally interrupt-safe; therefore, they should not be called directly or indirectly from an interrupt service routine.

The compiler uses the 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` should therefore include the `stdio.h` header file, especially if it is compiled with the -double-size-{32 | 64} switch . Failure to include the header file may result in a linker failure as the compiler must see a correct function prototype in order to generate the correct calling sequence.

This release provides three alternative run-time libraries that implement the functionality of the header file. If an application is built with the -full-io switch, then it is linked with a third-party I/O library that provides a comprehensive implementation of the ANSI C Standard I/O functionality, but at the cost of performance. It also supports printing of the native fixed-point types `fract` and `accum` in decimal format. No source files are provided for this proprietary library.

However, the normal behavior of the compiler is to link an application against an I/O library provided by Analog Devices-this library does not support all the facilities of the third-party library, but it is both faster and smaller. To reduce the size of the library, the native fixed-point types `fract` and `accum` are only printed in hexadecimal format. The source files for this library are available under the CCES installation in the subdirectory `Blackfin\lib\src\libio`.

A third option is to link an application against a variant of this default I/O library containing extra support for printing the native fixed-point types `fract` and `accum` in decimal format. You can do this by building the application with the -fixed-point-io switch. As before, this library does not support all the facilities of the third-party library, but it is both faster and smaller. The source files for this library are available under the CCES installation in the subdirectory `Blackfin\lib\src\libio`.

At program termination, any output that is pending in an I/O buffer is flushed to the appropriate stream and the host environment will then close down any physical connection between the application and an opened file. Note, however, that the I/O library does not implicitly close any opened streams to avoid unnecessary overheads (particularly with respect to memory occupancy); this means, for example, that any heap space used for file tables or I/O buffers will not be freed unless the associated stream is explicitly closed by the application.

The functional differences between the library based on third-party software (and accessed via the `-full-io` switch) and the default I/O run-time library provided by Analog Devices are given below:

- The third-party I/O library supports the input and output of wide characters (data of type `wchar_t`) and multi-byte characters. No similar support is available under the Analog Devices I/O library.

- The `fread()` and `fwrite()` functions are commonly used to transmit data between an application and a binary stream. For efficiency, the Analog Devices I/O library may not use a buffer to read or write data using these functions; thus, the data may be transmitted directly between a program and an external device. If an application relies on these functions to read and write data via an I/O buffer, it should be linked against the third-party library (using the `-full-io` switch).

- The functions `tmpfile` and `tmpnam` are only supported by the third-party I/O library, albeit with limited functionality; refer to the reference page for each of these functions for more details.

- When inputting formatted data (via fscanf, sscanf, and so on), both the third-party I/O library and the default I/O library support the following additional size qualifiers, which are defined in the C99 (ISO/IEC 9899:1999) standard.

  `hh signed char` *or* `unsigned char`

  `j intmax_t` *or* `uintmax_t`

  `t ptrdiff_t`

  `z size_t`

  These additional qualifiers may be used with the d, i, o, u, x, or X conversion specifiers to describe the type of the corresponding argument. However, only the third-party I/O library also supports these additional size qualifiers when printing formatted data using printf and its associated functions.

- The third-party I/O library accesses the current locale to determine the symbol to be used as the decimal point character.

- The third-party I/O library accepts the values `nan` and `inf` (in any case) as input for the e, f, and g conversion specifiers, these values represent the IEEE floating-point values for `NaN` (Not-A-Number) and Infinity respectively.

- The form of the output generated for the `a` conversion specifier by the alternative libraries differ (both forms of output do, however, conform to the requirements of ISO/IEC 9899:1999).

- The conversion specifier `F` is accepted by the third-party I/O library; the specifier behaves the same as `f`.

- The third-party I/O library also supports the full functionality of the [conversion specifier, while the Analog Devices I/O library only provides the minimum facility level required by the ANSI standard.

The implementation of both I/O libraries is based on a simple interface provided by the CCES simulator and EZ-KIT Lite® systems; for further details of this interface, refer to the *System Run-Time Documentation*.

Applications should be aware that this interface is activated under any of the following conditions:

- When a file is opened or closed

- When an input buffer becomes empty, or an output buffer becomes full or is flushed

- When interrogating or re-positioning a file pointer

- When deleting a file, via the remove library function

- When renaming a file, via the rename library function

Under all the above conditions, the interface will disable interrupts, and will halt the processor while it negotiates with the host to perform the required I/O operation. Once the I/O operation has completed, the interface will re-start the processor and then re-enable interrupts.

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

The following restrictions apply to either library in this software release:

- 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 type long double is only supported when an application is built with the -double-size-64 switch.

stdlib.h

The stdlib.h header file (see the *Library Functions in stdlib.h Header File* table in Documented Library Functions) offers general utilities specified by the C standard. These include 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.

string.h

The string.h header file (see the *Library Functions in string.h Header File* table in Documented Library Functions) contains string handling functions, including strcpy and memcpy.

sys/adi_core.h

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

time.h

The `time.h` header file (see the ***Library Functions in time.h Header File*** table in Documented Library Functions) provides functions, data types, and a macro for expressing and manipulating date and time information. The header file defines two fundamental data types: `clock_t` and `time_t`.

The `clock_t` data type is associated with the number of implementation-dependent processor "ticks" used since an arbitrary starting point.

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 *calendar time*. In this implementation, the epoch starts on the 1st of 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            */
};
```

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

The `time.h` header file sets the `CLOCKS_PER_SEC` macro to the number of processor cycles per second. 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.

> **CAUTION:** Generally, processor speed is a property of a particular processor. Therefore, it is recommended that the value to which this macro is set be verified independently before being used by an application.

By default, the value of the `CLOCKS_PER_SEC` macro is defined by the header file `cycles.h`. You may override this value by one of the following methods (listed in descending order of precedence):

- Via the `-DCLOCKS_PER_SEC=<definition>` compile-time switch. Because the `time_t` type is based on the `long long int` data type, it is recommended that the value of the symbolic name `CLOCKS_PER_SEC` be defined to be of type `long long int` by qualifying the value with the `LL` (or `ll`) suffix. For example, `-DCLOCKS_PER_SEC=6000000LL`

- Via the System Services Library

- Via the *Processor speed* option, found at *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor*.

Calling a Library Function From an ISR

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

- It must not update any global data, such as `errno`, and

- It must not write to (or maintain) any private static data

It is recommended that none of the functions defined in the `math.h` header file, nor the string conversion functions defined in the `stdlib.h` header file, 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 (such as `malloc`, `calloc`, `realloc`, and `free`), the C++ operators `new` and `delete`, and any variants, read and update global tables and are not interrupt-safe; they should not be called from an ISR. The heap debugging library can detect calls to memory allocation routines from an ISR, see Heap Debugging in the *Achieving Optimal Performance From C/C++ Source Code* chapter. for more information.

The following library functions are not interrupt-safe because they use private static data.

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

While not all C run-time library functions are interrupt-safe, *thread-safe* versions of the functions are available for use in a multi-threaded environment. These library functions are found in the run-time libraries that have an `mt` suffix in their file names.

C++ Library Support

By default, the compiler will use header files and functions specified by the abridged C++ library. If the -no-full-cpplib switch is enabled, the compiler uses header files and functions specified by the ISO/IEC 14882:2003 C++ standard.

The abridged C++ library has two major components: the embedded C++ library (EC++), and the standard template library (STL) as defined by the ISO/IEC 14882:2003 C++ standard. 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 in the CCES online help.

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

- Embedded C++ Library Header Files

- Standard C++ Library Header Files

- Common Standard and Embedded C++ Library Header Files

- C++ Header Files for C Library Facilities

- Standard Template Library (STL) Header Files

Embedded C++ Library Header Files

The *Embedded C++ Library Header Files* table describes the header files specifically implemented for the abridged C++ library.

Table 4-14: Embedded C++ Library Header Files

Header	Description
complex	Defines a template class `complex` and a set of associated arithmetic operators. Predefined types include `complex_float` and `complex_long_double`. The embedded 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 file and the C library header file `complex.h` refer to two different and incompatible implementations of the `complex` data type.
fstream	Defines the `filebuf`, `ifstream`, and `ofstream` classes for external file manipulations.
iomanip	Declares several `iostream` manipulators. Each manipulator accepts a single argument.
ios	Defines several classes and functions for basic `iostream` manipulations. Note that most of the `iostream` header files include `ios`.
iosfwd	Declares forward references to various `iostream` classes defined in other standard headers.
iostream	Declares most of the `iostream` objects used for the standard stream manipulations.
istream	Defines the `istream` class for `iostream` extractions. Note that most of the `iostream` header files include `istream`.
ostream	Defines the `ostream` class for `iostream` insertions.
sstream	Defines the `stringbuf`, `istringstream`, and `ostringstream` classes for various `string` object manipulations.
streambuf	Defines the `streambuf` classes for basic operations of the `iostream` classes. Note that most of the `iostream` header files include `streambuf`.
string	Defines a number of functions that help you manipulate C strings and other array of characters.
strstream	Defines the `strstreambuf`, `istrstream`, and `ostream` classes for `iostream` manipulations on allocated, extended, and freed character sequences.

Standard C++ Library Header Files

The *Standard C++ Library Header Files* table describes the header files that are included by the ISO/IEC 14882:2003 C++ standard library.

Table 4-15: Standard C++ Library Header Files

Header	Description
bitset	Defines a template class `bitset` and two supporting templates.

Table 4-15: Standard C++ Library Header Files (Continued)

Header	Description
complex	Defines a template class complex and a host of supporting template functions.
fstream	Defines several types and functions that support iostreams operations on sequences stored in external files.
iomanip	Declares several iostream manipulators. Each manipulator accepts a single argument.
ios	Defines several classes and functions for basic iostream manipulations. Note that most of the iostream header files include ios.
iosfwd	Declares forward references to various iostream template classes defined in other standard headers.
iostream	Declares objects that control reading and writing to the standard streams. It is often the only header required to perform input and output.
istream	Defines the template class basic_istream, which manipulates extractions for the iostreams.
limits	Defines the template class numeric_limits.
locale	Defines a number of template classes and functions that manipulate and encapsulate locales.
ostream	Defines the template class basic_ostream, which mediates insertions for the iostreams.
sstream	Defines a number of template classes that support iostream operations on sequences stored in an allocated array object.
streambuf	Defines the template class basic_streambuf, which is basic to the operation of the iostreams classes.
string	Defines the container class basic_string and various supporting templates.
strstream	Defines several classes that support iostreams operations on sequences stored in an allocated array of char object.
valarray	Defines the template class valarray and a number of supporting template classes and functions.

Common Standard and Embedded C++ Library Header Files

The *Common C++ Library Header Files* table describes the header files that are common to and included by both the ISO/IEC 14882:2003 C++ standard library and the abridged C++ library.

Table 4-16: Common C++ Library Header Files

Header	Description
exception	Defines several types and functions related to the handling of exceptions.
new	Declares several classes and functions for memory allocations and deallocations.
stdexcept	Defines a variety of classes for exception reporting.
typeinfo	Defines several types associated wit the type identification operator typeid, which yields information about both static and dynamic types.

C++ Header Files for C Library Facilities

For each C standard library header, there is a corresponding standard C++ header. For example, if the name of a C standard library header file were `foo.h`, the equivalent C++ header file would be named `cfoo`. Thus, 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 header files may be used to define names in the C++ global namespace, and the equivalent C++ header files define names in the standard namespace.

Table 4-17: C++ Header Files for C Library Facilities

Header	Description
cassert	Enforces assertions during function executions
cctype	Classifies characters
cerrno	Tests error codes reported by library functions
cfloat	Tests floating-point type properties
climits	Tests integer type properties
clocale	Adapts to different cultural conventions
cmath	Provides common mathematical operations
csetjmp	Executes non-local `goto` statements
csignal	Controls various exceptional conditions
cstdarg	Accesses a various number of arguments
cstddef	Defines several useful data types and macros
cstdio	Performs input and output
cstdlib	Performs a variety of operations
cstring	Manipulates several kinds of strings
cwchar	Manipulates wide strings. This is implemented for the full standard library only. (See the -full-cpplib switch in the *Compiler* chapter.)
cwtype	Classifies and maps codes for the target wide-character set. This is implemented for the full standard library only. See the -full-cpplib switch in the *Compiler* chapter.

Standard Template Library (STL) Header Files

Templates and the associated header files as defined by the ISO/IEC 14882:2003 C++ standard are not part of the embedded C++ standard library, but are supported by the compiler in C++ mode. The *Standard Template Library (STL) Header Files* table describes the standard template library header files.

The embedded C++ library and the standard C++ library also include several headers for compatibility with traditional C++ libraries; see the *Header Library Files for Compatibility with Traditional C++ Libraries* table.

Table 4-18: Standard Template Library (STL) Header Files

Header	Description
algorithm	Defines numerous common operations on sequences
deque	Defines a deque template container
functional	Defines numerous function templates that can be used to create callable types
hash_map	Defines two hashed map template containers.
hash_set	Defines two hashed set template containers
iterator	Defines common iterators and operations on iterators
list	Defines a list template container
map	Defines two map template containers
memory	Defines facilities for managing memory
numeric	Defines several numeric operations on sequences
queue	Defines two queue template container adapters
set	Defines two set template containers
stack	Defines a stack template container adapter
utility	Defines an assortment of utility templates
vector	Defines a vector template container

Table 4-19: Header Library Files for Compatibility with Traditional C++ Libraries

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

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 use 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 provided by the compiler.

Fatal Error Handling

The CCES run-time library provides a global mechanism for handling non-recoverable, or fatal, errors that are encountered during the execution of an application. This is provided by the functions adi_fatal_error and

adi_fatal_exception, which will write information related to the encountered error before looping around the break-points `__fatal_error` and `__fatal_exception`, respectively.

Four 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 was reported

- A value related to the error. This may not be relevant and may be left empty.

These functions will disable interrupts on entry, to prevent any code running whilst in a unknown state.

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 to which it is associated.

Table 4-20: 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

`FatalError.xml`, contained in the `System` directory of your 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

Three general codes are associated with the run-time libraries, `LibraryError`, `RunTimeError`, and `UnhandledException` which refer to errors identified with the use of the run-time libraries, errors associated with run-time environment, and exceptions which don't have a handler set up, respectively. 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 4-21: General Error Codes Used by Run-Time Library

General Code	Name	Value
Run-time library error	LibraryError	0x7
Run-time environment error	RunTimeError	0x8
Unhandled exception	UnhandledException	0x9
Parity errors	ParityError	0xA
errno values	Errno	0xB
User-defined error	UserError	0xffffffff

Specific Codes

This section lists and describes the specific library and run-time errors that may occur.

Library Errors

Specific code list, associated with the LibraryError general code, details any fatal errors identifiable by use of the run-time libraries. These errors are described in the *LibraryError Specific Codes* table.

Table 4-22: LibraryError 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	OSALBindingError	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 4-22: LibraryError Specific Codes (Continued)

Specific Code Value	Error	Description	Error Value Interpretation
0x108	HeapFunctionMisMatch	A heap operation which is 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 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 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 an `.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

Run-Time Errors

Specific codes, associated with the `RunTimeError` general code, detail any fatal errors identifiable by the use of the run-time environment. These errors are described in the ***RunTimeError Specific Codes*** table.

Table 4-23: RunTimeError Specific Codes

Specific Code Value	Error	Description	Error Value Interpretation
0x1	CPLBMissAllLocked	A CPLB miss has occurred where all active CPLBs are locked	None
0x2	CPLBMissWithoutReplacement	A CPLB miss has occurred without a corresponding CPLB entry	None
0x3	CPLBProtectionViolation	A CPLB protection violation has occurred	None
0x4	CPLBAddressIsMisalignedForCPLB	A CPLB address is misaligned for the size of that CPLB	None
0x5	L1CodeCacheEnabledWhenL1UsedForCode	L1 code cache has been enabled when L1 is used for code	None
0x6	L1DataACacheEnabledWhenUsedForData	L1 dataA cache is enabled when it is used for data	None
0x7	L1DataBCacheEnabledWhenUsedForData	L1 dataB cache is enabled when it is used for data	None
0x8	TooManyLockedDataCPLB	Too many data CPLBs have been locked	None
0x9	TooManyLockedInstructionCPLB	Too many instruction CPLBs have been locked	None
0xB	NoDispatchedHandler	No dispatched handler available for the specified interrupt code.	None
0x100	DMASrcConfigErr	DMA has failed due to an error with DMA source configuration.	None
0x101	DMASrcError	DMA has failed due to an error with DMA source.	None
0x110	DMASrcIllegalWrite	DMA has failed due to an illegal write in source DMA.	None
0x120	DMASrcAlignment	DMA has failed due to an alignment write in source DMA.	None
0x130	DMASrcMemErr	DMA has failed due to a memory/fabric error in source DMA.	None
0x150	DMASrcTriggerOverrun	DMA has failed due to trigger overrun in source DMA.	None
0x160	DMASrcBWMon	DMA has failed due to a bandwidth monitor in source DMA.	None
0x200	DMADstConfigErr	DMA has failed due to an error with DMA destination configuration.	None
0x201	DMADstError	DMA has failed due to an error with DMA destination.	None
0x202	DMADstPatternSizeInvalid	DMA pattern length invalid.	None
0x210	DMADstIllegalWrite	DMA has failed due to an illegal write in destination DMA.	None
0x220	DMADstAlignment	DMA has failed due to an alignment write in destination DMA.	None
0x230	DMADstMemErr	DMA has failed due to a memory/fabric error in destination DMA.	None
0x250	DMADstTriggerOverrun	DMA has failed due to trigger overrun in destination DMA.	None
0x260	DMADstBWMon	DMA has failed due to a bandwidth monitor in destination DMA.	None
0x301	UnexpectedCPLBMgrReturn	An unexpected value has been returned by the CPLB Manager.	None
0x310	DCPLBMissAllLocked	A data CPLB miss has occurred where all active CPLBs are locked.	None

Table 4-23: RunTimeError Specific Codes (Continued)

Specific Code Value	Error	Description	Error Value Interpretation
0x311	ICPLBMissAllLocked	An instruction CPLB miss has occurred where all active CPLBs are locked.	None
0x320	DCPLBMissWithoutReplacement	A data CPLB miss has occurred without a corresponding CPLB entry.	None
0x321	ICPLBMissWithoutReplacement	An instruction CPLB miss has occurred without a corresponding CPLB entry.	None
0x330	DCPLBProtectionViolation	An illegal data memory access has occurred.	None
0x331	ICPLBProtectionViolation	An illegal instruction memory access has occurred.	None
0x340	DCPLBDoubleHit	More than one data CPLB covers the accessed location.	None
0x341	ICPLBDoubleHit	More than one instruction CPLB covers the accessed location.	None

Unhandled Exceptions

Specific codes, associated with the `UnhandledException` general code, detail any exceptions that do not have a handler set. These exceptions are described in the ***UnhandledException Specific Codes*** table.

Table 4-24: UnhandledException Specific Codes

Specific Code Value	Error	Description	Error Value Interpretation
0x11	TraceBufferFull	The trace buffer has overflowed	None
0x21	UndefinedInstruction	An undefined instruction has been encountered	None
0x22	IllegalInstructionCombination	An illegal instruction combination has been encountered	None
0x23	DataAccessCPLBProtection	Attempted read/write of supervisor resource, or illegal data access	None
0x24	DataMisalignedAccessViolation	Attempted misaligned data access	None
0x25	UnrecoverableEvent	An unrecoverable event has occurred	None
0x26	DataCPLBMiss	CPLB miss on data fetch	None
0x27	DataCPLBMultipleHits	Multiple CPLBs match data fetch address	None
0x28	EmulationWatchpoint	There is a watchpoint match	None
0x2A	InstructionFetchMisaligned	Attempted misaligned instruction cache fetch	None
0x2B	InstructionFetchViolation	Illegal instruction fetch access	None
0x2C	InstructionCPLBMiss	CPLB miss on instruction fetch	None
0x2D	InstructionCPLBMultipleHits	Multiple CPLBs match instruction fetch address	None

Table 4-24: UnhandledException Specific Codes (Continued)

Specific Code Value	Error	Description	Error Value Interpretation
0x2E	SupervisorResource	Attempt to use supervisor resource from user mode	None

Parity Errors

Specific codes, associated with the `ParityError` general code, describe parity errors in L1 memory. Two kinds of error depend on whether the location of the parity error can be identified. Where the location is known, the specific code is a three-digit code `0xXYZ`, where:

- `X` indicates instruction memory (1) or data memory (2).

- `Y` indicates the bank: A (1), B (2), or C (3).

- `Z` indicates the memory type: SRAM (1), cache tags (2), or scratchpad (4).

The specific list in `FatalError.xml` contains several values, which are combined to produce the errors show in the *ParityError Specific Codes* table; the individual values are never produced by themselves, so are not listed here. The combined errors codes are listed in the *ParityError Specific Codes* table.

Table 4-25: ParityError Specific Codes

Specific Code Value	Error	Description	Error Value Interpretation
0x1	NonSpeculativeAccessAborted	A non-speculative access has been aborted due to L1 parity error.	None
0x2	InstrReadForL2	Parity error on instruction L1 read for L2 transfer	None
0x3	DataReadForL2	Parity error on data L1 read for L2 transfer	None
0x110	InstrBankA	Parity error in L1 instruction bank A SRAM	None
0x120	InstrBankB	Parity error in L1 instruction bank B SRAM	None
0x141	InstrBankCSRAM	Parity error in L1 instruction bank C SRAM	None
0x142	InstrBankCCache	Parity error in L1 instruction bank C cache	None
0x211	DataBankASRAM	Parity error in L1 data bank A SRAM	None
0x212	DataBankACache	Parity error in L1 data bank A cache	None
0x213	DataBankAXPAD	Parity error in L1 data bank A scratchpad	None
0x221	DataBankBSRAM	Parity error in L1 data bank B SRAM	None
0x222	DataBankBCache	Parity error in L1 data bank B cache	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, 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 4-26: Library Functions in the adi_libldr.h Header File

adi_libldr_Load	adi_libldr_LoadDbg	adi_libldr_Move
adi_libldr_MoveDbg	adi_libldr_LookUpSym	

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

Table 4-27: ccblkfn.h Library Functions

adi_acquire_lock, adi_try_lock, adi_release_lock	adi_core_enable, adi_core_1_enable, adi_core_1_disable, adi_core_b_enable	adi_core_id
adi_obtain_mc_slot, adi_free_mc_slot, adi_set_mc_value, adi_get_mc_value	_l1_memcpy, _memcpy_l1	

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

Table 4-28: complex.h Library Functions that operate on C99 complex types

cabs	carg	cexp
cimag	conj	creal

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

Table 4-29: ctype.h Library Functions

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

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

Table 4-30: heap_debug.h Library Functions

adi_dump_all_heaps	adi_dump_heap
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 *instrprof.h Library Functions* table lists functions in the `instrprof.h` header file. For more information, see instrprof.h.

Table 4-31: instrprof.h Library Functions

instrprof_request_flush

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

ATTENTION: This support for dynamically-loadable modules (DLMs) using `libdyn.h` has been deprecated, and will be removed in a future release. Please see Dynamically-Loadable Modules in the System Runtime documentation for information on how to use the APIs in adi_libldr.h to load a DLM from a loader stream.

Table 4-32: libdyn.h Library Functions

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

Table 4-33: math.h Library Functions

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

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

Table 4-34: mc_data.h Library Functions

adi_obtain_mc_slot, adi_free_mc_slot, adi_set_mc_value, adi_get_mc_value	adi_obtain_mc_slot, adi_free_mc_slot, adi_set_mc_value, adi_get_mc_value	adi_obtain_mc_slot, adi_free_mc_slot, adi_set_mc_value, adi_get_mc_value	adi_obtain_mc_slot, adi_free_mc_slot, adi_set_mc_value, adi_get_mc_value

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

Table 4-35: pgo_hw.h Library Functions

pgo_hw_request_flush

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

Table 4-36: setjmp.h Library Functions

longjmp	setjmp

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

Table 4-37: signal.h Library Function

raise	signal

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

Table 4-38: stdarg.h Library Functions

va_arg	va_end	va_start

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

Table 4-39: stdfix.h Library Functions

absfx	bitsfx	countlsfx
divifx	fxbits	fxdivi
idivfx	mulifx	roundfx
strtofxfx		

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

Table 4-40: stdio.h Library Functions

clearerr	fclose	feof
ferror	fflush	fgetc
fgetpos	fgets	fileno
fopen	fprintf	fputc
fputs	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
tmpfile	tmpnam	vfprintf
vprintf	vsprintf	vsnprintf

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

Table 4-41: stdlib.h Library Functions

abort	abs	adi_fatal_error
adi_fatal_exception	atexit	atof
atoi	atol	calloc
atoll	bsearch	free
div	exit	heap_init
heap_calloc	heap_free	heap_malloc
heap_install	heap_lookup	labs
heap_realloc	heap_space_unused	qsort
ldiv	malloc	space_unused
rand	realloc	strtof
srand	strtod	strtoll
strtol	strtoul	strtold
strtoull		

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

Table 4-42: string.h Library Functions

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

The ***sys/adi_core.h Library Functions*** table lists functions in the `sys/adi_core.h` header file. This header file is included by `ccblkfn.h`. For more information, see sys/adi_core.h.

Table 4-43: sys/adi_core.h Library Functions

adi_core_enable, adi_core_1_enable, adi_core_1_disable, adi_core_b_enable	adi_core_enable, adi_core_1_enable, adi_core_1_disable, adi_core_b_enable	adi_core_id

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

Table 4-44: time.h Library Functions

asctime	clock	ctime
difftime	gmtime	localtime
mktime	strftime	time

C Run-Time Library Reference

The C run-time library is a collection of functions called from your C programs. The following items apply to all of the functions in the library.

Notation Conventions

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

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

abort

Abnormal program end

Synopsis

```
#include <stdlib.h>

void abort(void);
```

Description

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

Error Conditions

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 input.

> **NOTE:** The result of abs(INT_MIN) is undefined.

Error Conditions

None

Example

```
#include <stdlib.h>

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

See Also

absfx, fabs, labs

absfx

Absolute value

Synopsis

```
#include <stdfix.h>

short fract abshr(short fract f);
fract absr(fract f);
long fract abslr(long fract f);
short accum abshk(short accum a);
accum absk(accum a);
long accum abslk(long accum a);
```

Description

The absfx family of functions returns 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. Use the macro with any of the fixed-point types to return a result of the same type as its operand.

Error Conditions

None

Example

```
#include <stdfix.h>

accum a;
long fract f;

a = abshk(-12.5k);          /* a == 12.5k */
a = absfx(-12.5k);          /* a == 12.5k */
f = abslr(0.75lr);          /* f == 0.75lr */
f = absfx(0.75lr);          /* f == 0.75lr */
```

See Also

abs, fabs, labs

acos

Arc cosine

Synopsis

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

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

fract16 acos_fr16 (fract16 x);
fract32 acos_fr32 (fract32 x);

_Fract acos_fx16 (_Fract x);
long _Fract acos_fx32 (long _Fract x);
```

Description

The arc cosine functions return the arc cosine of x. Both the argument x and function results are in radians.

The input for acos, acosf, and acosd must be in the range [-1, 1]. The functions return a result in the range [0, π].

The acos_fr16, acos_fr32, acos_fx16, and acos_fx32 functions are defined for fractional input values between 0 and 0.9. The outputs from the functions are in the range [acos(0)*2/π, acos(0.9)*2/π].

> **NOTE:** The acosf, acosd, acos_fr16 and acos_fr32 functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

If the input is not in the defined range, the arc cosine functions return a zero.

Example

```
#include <math.h>

double y;
y = acos(0.0);        /* y = PI/2 */
```

See Also

cos

adi_acquire_lock, adi_try_lock, adi_release_lock

Obtain and release locks for multi-core synchronization

Synopsis

```
#include <ccblkfn.h>

void adi_acquire_lock(testset_t *lockptr);
int adi_try_lock(testset_t *lockptr);
void adi_release_lock(testset_t *lockptr);
```

Description

These functions provide locking facilities for multi-core applications that need to ensure private access to shared resources, or for applications that need to build synchronization mechanisms.

The functions operate on a pointer to a `testset_t` object, which is a private type used only by these routines. Objects of type `testset_t` must be global, and initialized to zero (which indicates that the lock is unclaimed). The type is automatically volatile.

The `adi_acquire_lock` function repeatedly attempts to acquire the lock, until successful. Upon return, the lock will have been acquired. The function does not make use of any timers or other mechanisms to pause between attempts, so this function implies continuous accesses to the lock object.

The `adi_try_lock` function makes a single attempt to acquire the lock, but does not block if the lock has already been acquired. The function returns non-zero if it has successfully acquired the lock, and zero if the lock was not available.

The `adi_release_lock` function releases the lock object, marking it as available to the next attempt by `adi_acquire_lock` or `adi_try_lock`. The `adi_release_lock` function does not return a value, and does not verify whether the caller already holds the lock, or even if the lock is already held by "another" caller.

NOTE: • `TESTSET` instruction silicon errata

• There are various silicon anomalies related to the `TESTSET` instructions used by these functions. You should review the CCES silicon anomaly documentation and the Silicon errata sheet for your part to be sure these functions are safe to use.

Error Conditions

These functions do not return error conditions. Neither `adi_acquire_lock()` nor `adi_release_lock()` return values. The `adi_try_lock` function merely returns a value indicating whether the lock was acquired.

Examples

```
#include <ccblkfn.h>

void add_one(testset_t *lockptr, volatile int *valptr)
{
   adi_acquire_lock(lockptr);
   *valptr += 1;
   adi_release_lock(lockptr);
}
```

NOTE: To be useful, the `testset_t` object must be located in a shared area of memory accessible by both cores. These functions do not disable interrupts; that is the responsibility of the caller.

```
#include <ccblkfn.h>
```

```
void claim_lock(testset_t *lockptr)
{
    while (!adi_try_lock(lockptr)) {
        // do something else or go to sleep
        // before trying the lock again
    }
}
```

See Also

adi_core_id, adi_obtain_mc_slot, adi_free_mc_slot, adi_set_mc_value, adi_get_mc_value

adi_core_enable, adi_core_1_enable, adi_core_1_disable, adi_core_b_enable

Enable or disable another core

Synopsis

```
#include <ccblkfn.h>

ADI_CORE_ENABLE_STATUS adi_core_enable(ADI_CORE_ID id);
void adi_core_1_enable(void);
void adi_core_1_disable(void);
void adi_core_b_enable(void);
```

Description

The `adi_core_enable`, `adi_core_x_enable` functions are available on multi-core processors, to release the other available cores. Due to differences in processor terminology, the ADSP-BF561 variant is called `adi_core_b_enable`, while the other functions identify processors numerically from zero (e.g. `adi_core_1_enable`). The `adi_core_enable` function works on all multi-core processors. Once released, the core being executing instructions from its reset address.

The emulator releases additional cores as part of the program-loading process, so startup code employs a software lock to ensure that additional cores do not start running their applications too soon; the `adi_core_x_enable` functions toggle the software lock as well as releasing the core, allowing any emulator-released cores to continue.

The `adi_core_enable` function takes a parameter of the enumeration type `ADI_CORE_ID` as the core ID of the core to enable. Since the function is intended to be called from core 0, the only valid parameter is `ADI_CORE_1`, which corresponds to core b on ADSP-BF561 parts and core 1 on other multi-core Blackfin parts. On success, the function returns `ADI_CORE_ENABLE_SUCCESS`.

Where available, the `adi_core_x_disable` function puts the specified function back into reset mode.

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 available on the part.

- `ADI_CORE_ENABLE_CURRENT_CORE`: The core ID parameter specified the same core which is being used to execute the function.

- `ADI_CORE_ENABLE_ALREADY_ENABLED` The core ID parameter specified a core which could not be enabled as it is already running.

The other functions have no error conditions.

Example

```
#include <ccblkfn.h>

int main(void)
{
    // Core 1 is in Reset
       adi_core_1_enable();
    // Core 1 is now running
}
```

See Also

adi_acquire_lock, adi_try_lock, adi_release_lock, adi_core_id

adi_core_id

Identify caller's core

Synopsis

```
#include <ccblkfn.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_0` value when executed by core 0 or core A, and a value of `ADI_CORE_1` when executed on core 1 or core B.

Error Conditions

None

Example

```
#include <ccblkfn.h>

const char *core_name(void)
{
   if (adi_core_id() == ADI_CORE_0)
      return "Core A";
   else
      return "Core B";
}
```

See Also

adi_acquire_lock, adi_try_lock, adi_release_lock, adi_obtain_mc_slot, adi_free_mc_slot, adi_set_mc_value, adi_get_mc_value, adi_obtain_mc_slot, adi_free_mc_slot, adi_set_mc_value, adi_get_mc_value

adi_dump_all_heaps

Dump the current state of all 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 which have been tracked by the heap debugging library, and the total memory currently allocated from that heap.

If the specified file exists, the file will be 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) is not included in the output.

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

For more information, see Heap Debugging.

Error Conditions

`adi_dump_heap` 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 which have been tracked by the heap debugging library, and the total memory currently allocated from that heap.

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

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

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) is not included in the output.

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

For more information, see Heap Debugging.

Error Conditions

`adi_dump_heap` returns false if the heap specified by `heapindex` does not exist.

`adi_dump_heap` 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` are written to global variables along with the return address, before looping around the label `__fatal_error`.

Interrupts are disabled by adi_fatal_error since safe execution cannot be relied on.

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`.

Interrupts are disabled by `adi_fatal_exception` since safe execution cannot be relied on.

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 the 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, see Heap Debugging.

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 the features are to be enabled. These bits are represented by macros defined in heap_debug.h. _HEAP_TRACK_USAGE (track heap activity) will be 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, see Heap Debugging.

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. `adi_heap_debug_end` 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, see Heap Debugging.

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:** Call `adi_heap_debug_flush` only 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, see Heap Debugging.

Error Conditions

`adi_heap_debug_flush` 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_resume

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, see Heap Debugging.

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

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

Table 4-45: Reset Values for Heap Guard Regions

Region	*Value*
Free block	`0xBD`
Allocated block	`0xDD`
Block content (not `calloc`)	`0xED`

For more information, see Heap Debugging.

Error Conditions

`adi_heap_debug_reset_guard_region` 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 deallocations which 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, see Heap Debugging.

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, see Heap Debugging.

> **NOTE:** Call `adi_heap_debug_set_buffer` only 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

`adi_heap_debug_set_buffer` returns false if the buffer passed is not valid or big enough to be used by 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);
}
```

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 eight bytes per call stack element. The default value is five stack elements deep.

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

For more information, see Heap Debugging.

Error Conditions

`adi_heap_debug_set_call_stack_depth` 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;
}
```

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 will print a diagnostic message to stderr before calling `_adi_fatal_error`.

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

For more information, see Heap Debugging.

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

Change 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 will be checked for guard region corruption before changing the guard regions. If any guard region is corrupt,

`adi_heap_debug_set_guard_region` will fail and the guard regions will not be changed. The contents of existing allocations will not be changed, but any new allocations will be pre-filled with the pattern specified by the `allocated` parameter.

The value of `free` will be written to any blocks which are free, as well as the preceding guard region. Corruption of these blocks indicates that a pointer that has been written to after it has been freed.

The value of `allocated` will be 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` will be written to the allocated memory block, with the exception of memory allocated by `calloc`, which will be 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`, respectively. These variables can be defined at build-time but should not be written to directly at runtime, or false corruption errors may be reported.

The guard region values can be reset to the Analog Devices default values by calling `adi_heap_debug_reset_guard_region`.

For more information, see Heap Debugging.

Error Conditions

`adi_heap_debug_set_guard_region` 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(0x11, 0x22, 0x33)) {
        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 will produce no run-time diagnostics, but will appear in the heap debugging report (if generated).

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

For more information, see Heap Debugging.

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

Any detected errors will be 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, 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 will have no effect.

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

For more information, see Heap Debugging.

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 deallocating and reallocating
   // pointers not returned by an allocation function and
   // about deallocations 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.
ADI_LIBLDR_MEMCPY_L1_ ERROR	The `_memcpy_l1` function failed to copy a code section from the loader stream to a target address in L1 memory. This could occur if the target address is not aligned to an 8-byte boundary, or if the block length is not a multiple of 8 bytes.
ADI_LIBLDR_MALLOC_FAILED	`malloc` failed to allocate memory for a temporary buffer.

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 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.
`ADI_LIBLDR_MEMCPY_L1_ ERROR`	The `_memcpy_l1` function failed to copy a code section from the loader stream to a target address in L1 memory. This could occur if the target address is not aligned to an 8-byte boundary, or if the block length is not a multiple of 8 bytes.
`ADI_LIBLDR_MALLOC_FAILED`	`malloc` failed to allocate memory for a temporary buffer.

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_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_Load` 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.
ADI_LIBLDR_MEMCPY_L1_ ERROR	The `_memcpy_l1` function failed to copy a code section from the loader stream to a target address in L1 memory. This could occur if the target address is not aligned to an 8-byte boundary, or if the block length is not a multiple of 8 bytes.
ADI_LIBLDR_MALLOC_FAILED	`malloc` failed to allocate memory for a temporary buffer.

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 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.
ADI_LIBLDR_MEMCPY_L1_ ERROR	The _memcpy_l1 function failed to copy a code section from the loader stream to a target address in L1 memory. This could occur if the target address is not aligned to an 8-byte boundary, or if the block length is not a multiple of 8 bytes.
ADI_LIBLDR_MALLOC_FAILED	malloc failed to allocate memory for a temporary buffer.

Example

```
#include <adi_libldr.h>

extern char_t *ldr_stream;

/* read a loader stream from memory 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_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.

Returned Value	Reason
`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_obtain_mc_slot, adi_free_mc_slot, adi_set_mc_value, adi_get_mc_value

Obtain and manage storage for multi-core private data in shared functions

Synopsis

```
#include <mc_data.h>

int adi_obtain_mc_slot(int *slotID, void (fn)(void *));
int adi_free_mc_slot(int slotID);
int adi_set_mc_value(int slotID, void *valptr);
void *adi_get_mc_value(int slotID);
```

Description

These functions provide a framework for shared functions that may be called from any core in a multi-core environment, yet need to maintain data values that are private to the calling core. An example is `errno`-in a multi-core environment, each core needs to maintain its own version of the errno value, but the correct version of `errno` must be updated when a shared standard library function is called.

The framework operates by maintaining a set of "slots", each slot corresponds to a data object that must be core-local. The slot holds a pointer for each core, which can be set to point to the core's private version of the data object.

The process is as follows:

1. If this is the first time any core has needed the private data, allocate a slot.

2. If this is the first time this core has needed the private data, allocate storage for the data and record it in the slot. Otherwise, retrieve the location of the data's storage from the slot.

3. Access the data.

The `adi_obtain_mc_slot` function is called to allocate a slot, when no core has previously needed to access the data. `slotID` must be a pointer to a global variable, shared by all the cores, which is initialized to the value `adi_mc_unallocated`. The `fn` parameter must be NULL.

If the `adi_obtain_mc_slot` function can allocate a slot for the data object, it returns the slot's identifier, via the `slotID` pointer, and returns a non-zero value. If there are no more slots remaining, the function returns a zero value.

The `adi_free_mc_slot` function releases the slot indicated by `slotID`, which must have been previously allocated by the `adi_obtain_mc_slot` function. If `slotID` indicate a valid slot, the slot is freed and the function returns a non-zero value. If `slotID` does not indicate a currently-valid slot, the function returns zero.

The adi_set_mc_value function records the `valptr` pointer in the slot indicated by `slotID`, as the location of the private data object for the calling core. The function returns `1` if `slotID` refers to a currently-valid slot, otherwise the function returns `0`.

The `adi_obtain_mc_value` function returns a pointer previously stored in the slot indicated by `slotID`, for the calling core. The pointer must have been previously stored by the adi_set_mc_value function, by the current core, otherwise the function returns NULL. The function also returns NULL if `slotID` does not indicate a currently-valid slot.

Error Conditions

The `adi_obtain_mc_slot` function returns a zero value if a new slot cannot be allocated.

The adi_free_mc_slot and adi_set_mc_value functions both return a zero value if `slotID` does not refer to a currently-valid slot.

The `adi_obtain_mc_value` function returns NULL if `slotID` does not refer to a currently-valid slot, or if the calling core has not yet stored a pointer in the slot via `adi_set_mc_value`.

Example

```
/* error handling omitted */
#include <mc_data.h>
#include <ccblkfn.h>
#include <stdlib.h>

static int slotid = adi_mc_unallocated;
static testset_t slotlock = 0;

void set_error(int val)
```

```
{
   int *storage;
   adi_acquire_lock(&slotlock);
   if (slotid == adi_mc_unallocated) {
       // first core here
       adi_obtain_mc_slot(&slotid, NULL);
   }
   adi_release_lock(&slotlock);
   storage = adi_get_mc_value(slotid);
   if (storage == NULL)  {
      // first time this core is here
      storage = malloc(sizeof(int));
      adi_set_mc_value(slotid, storage);
   }
   *storage = val;
}
```

NOTE: The multi-core private storage routines do not disable interrupts; that is left at the caller's discretion.

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_heaps` 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 will be 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: `adi_verify_heap` 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 checked for corruption.

For more information, see Heap Debugging.

Error Conditions

`adi_verify_heap` returns false if any corrupt guard regions were 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

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 guard regions. If a corrupt guard region is detected on any heaps, `adi_verify_all_heaps` returns false; otherwise, true will be returned.

> **NOTE:** `adi_verify_all_heaps` 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 checked for corruption.

For more information, see Heap Debugging.

Error Conditions

`adi_verify_all_heaps` returns false if any corrupt guard regions were 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

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 contains 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);

fract16 asin_fr16(fract16 x);
fract32 asin_fr32(fract32 x);

_Fract asin_fx16(_Fract x);
long _Fract asin_fx32(long _Fract x);
```

Description

The arc sine functions return the arc sine of the argument x. Both the argument x and the function results are in radians.

The input for the functions `asin`, `asinf`, and `asind` must be in the range [-1, 1], and the functions return a result that will be the range [-π/2, π/2].

The `asin_fr16`, `asin_fr32`, `asin_fx16`, and `asin_fx32` functions are defined for fractional input values in the range [-0.9, 0.9]. The outputs from the functions are in the range [`asin(-0.9)*2/π`, `asin(0.9)*2/π`].

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The arc sine functions return a zero if the input is not in the defined range.

Example

```
#include <math.h>

double y;
y = asin(1.0);        /* y = PI/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);

fract16 atan_fr16 (fract16 x);
fract32 atan_fr32 (fract32 x);

_Fract atan_fx16 (_Fract x);
long _Fract atan_fx32 (long _Fract x);
```

Description

The arc tangent functions return the arc tangent of the argument. Both the argument `x` and the function results are in radians.

The `atanf`, `atan`, and `atand` functions return a result that is in the range [-π/2, π/2].

The `atan_fr16`, `atan_fr32`, `atan_fx16`, and `atan_fx32` functions are defined for fractional input values in the range [-1.0, 1.0). The outputs from the functions are in the range [-π/4, π/4].

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

None

Example

```
#include <math.h>

double y;
y = atan(0.0);        /* y = 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);
fract16 atan2_fr16 (fract16 y, fract16 x);
fract32 atan2_fr32 (fract32 y, fract32 x);
_Fract atan2_fx16 (_Fract y, _Fract x);
long _Fract atan2_fx32 (long _Fract y, long _Fract x);
```

Description

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

The `atan2f`, `atan2`, and `atan2d` functions return a result that is in the range [-π, π].

The `atan2_fr16`, `atan2_fr32`, `atan2_fx16`, and `atan2_fx32` functions are defined for fractional input values in the range [-1.0, 1.0). The outputs from these function are scaled by π and are in the range [-1.0, 1.0).

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 is 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 function reference `atof (pdata)` 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, 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 a zero if no conversion is made.

Example

```
#include <stdlib.h>

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

See Also

atof, atol, strtod, strtol, strtoul

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 a zero if no conversion is made.

Example

```
#include <stdlib.h>

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

See Also

atof, strtod, strtol, strtoul

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 whether a zero is a valid result or an indicator of an invalid string.

Error Conditions

The `atoll` function returns a zero if no conversion is made.

Example

```
#include <stdlib.h>

long long int i;
i = atoll("5");        /* i == 5 */
```

See Also

strtoll

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);
int_hk_t bitshk(short accum a);
int_k_t bitsk(accum a);
int_lk_t bitslk(long accum a);
uint_uhk_t bitsuhk(unsigned short accum a);
```

```
uint_uk_t bitsuk(unsigned accum a);
uint_ulk_t bitsulk(unsigned long accum a);
```

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>

int_k_t k;
uint_ulr_t ulr;
k = bitsk(-12.5k);              /* k == 0xfffffff9c0000000 */
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 calls 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` returns 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` returns `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 memory allocated is aligned to a 8-byte boundary.

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, 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.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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

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 in the *DSP Run-Time Library* chapter.

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);

fract16 cos_fr16 (fract16 x);
_Fract cos_fx16 (_Fract x);

fract32 cos_fr32 (fract32 x);
long _Fract cos_fx32 (long _Fract x);
```

Description

The cosine functions return the cosine of the argument. Both the argument `x` and the results returned by the functions are in radians.

The `cos_fr16`, `cos_fr32`, `cos_fx16` and `cos_fx32` functions input a fractional value in the range [-1.0, 1.0) corresponding to [-π/2, π/2]. The domain represents half a cycle which can be used to derive a full cycle if required (see *Notes* below). The result, in radians, is in the range [-1.0, 1.0).

The domain of `cosf` is [-102940.0, 102940.0], and the domain for `cosd` is [-843314852.0, 843314852.0]. The result returned by the functions `cos`, `cosf`, and `cosd` is in the range [-1, 1]. The functions return 0.0 if the input argument `x` is outside the respective domains.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

None

Example

```
#include <math.h>

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

Notes

The domain of the `cos_fr16`, `cos_fr32`, `cos_fx16` and `cos_fx32` functions is restricted to the range [-1, 1) which corresponds to half a period from -(π /2) to π/2. It is possible to derive the full period using the following properties of the function.

cosine [0, π/2] = -cosine [π, 3/2π]

cosine [-π/2, 0] = -cosine [π/2, π]

The function below uses these properties to calculate the full period (from 0 to 2π) of the cosine function using an input domain of [0, 0x7fff].

```
#include <math.h>

fract16 cos2pi_fr16 (fract16 x)
{
   if (x < 0x2000) {                          /* <0.25  */
      /* first quadrant [0..π/2):             */
      /* cos_fr16([0x0..0x7fff]) = [0..0x7fff)   */
      return cos_fr16(x * 4);

   } else if (x < 0x6000) {           /* < 0.75  */
      /* if (x < 0x4000)                       */
      /* second quadrant [π/2..π):             */
      /* -cos_fr16([0x8000..0x0)) = [0x7fff..0)  */
      /*                                       */

      /* if (x < 0x6000)                       */
      /* third quadrant [π..3/2π):             */
      /* -cos_fr16([0x0..0x7fff]) = [0..0x8000)  */
      return -cos_fr16((0xc000 + x) * 4);

   } else {
      /* fourth quadrant [3/2π..π):            */
      /* cos_fr16([0x8000..0x0)) = [0x8000..0)   */
      return cos_fr16((0x8000 + x) * 4);
   }
}
```

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.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The domain of `coshf` is [-87.33, 88.72], 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>

double x, y;
float v, w;

y = cosh (x);
v = coshf (w);
```

See Also

sinh

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);
int countlshk(short accum a);
int countlsk(accum a);
int countlslk(long accum a);
int countlsuhk(unsigned short accum a);
int countlsuk(unsigned accum a);
int countlsulk(unsigned long accum a);
```

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 returns 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 = countlsk(-12.5k);              /* n == 4 */
n = countlsfx(-12.5k);             /* n == 4 */
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

ldiv, divifx, fmod, fxdivi, 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);
int divik(int numer, accum denom);
long int divilk(long int numer, long accum denom);
unsigned int diviuk(unsigned int numer, unsigned accum denom);
unsigned long int diviulk(unsigned long int numer,
                          unsigned long accum 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` family of functions have undefined behavior if the denominator is zero.

Example

```
#include <stdfix.h>
```

```
int quo;
unsigned long int ulquo;
quo = divik(125, -12.5k);             /* quo == -10 */
ulquo = diviulr(125, 0.125ulr);       /* ulquo == 1000 */
```

See Also

fxdivi, idivfx

dyn_AddHeap

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

Returned Value	Reason
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. See ***Dynamically-Loadable Modules*** in the System Runtime 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 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 `dyn_GetHeapForWidth`.

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_ValidateImage, dyn_GetNumSections, dyn_GetSections, dyn_GetStringTableSize, dyn_GetStringTable, dyn_GetExpSymTab, dyn_LookupByName, dyn_Relocate, dyn_SetSectionAddr, dyn_CopySectionContents, 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. See ***Dynamically-Loadable Modules*** in the System Runtime 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 house-keeping 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_ValidateImage, dyn_GetNumSections, dyn_GetSections, dyn_GetStringTableSize, dyn_GetStringTable, dyn_GetExpSymTab, dyn_LookupByName, dyn_Relocate, dyn_SetSectionAddr, dyn_CopySectionContents, dyn_AllocSectionMem, dyn_FreeSectionMem, heap_malloc

dyn_CopySectionContents

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 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 copies 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 adi_libldr.h APIs 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 was 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. See *Dynamically-Loadable Modules* in the System Runtime 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 (DLM)

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.

For more information about DLMs, see *elf2dyn - ELF to Dynamically-Loadable Module Converter* in the *Loader and Utilities Manual*.

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. See ***Dynamically-Loadable Modules*** in the System Runtime 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>

DYN_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.

For more information about DLMs, see *elf2dyn - ELF to Dynamically-Loadable Module Converter* in the *Loader and Utilities Manual*.

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_AllocSectionMem, dyn_CopySectionContents

dyn_GetExpSymTab

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 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_AllocSectionMem, dyn_CopySectionContents

dyn_GetHeapForWidth

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 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 `dyn_AddHeap`), 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. See *Dynamically-Loadable Modules* in the System Runtime 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 relocations 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 contains 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. See ***Dynamically-Loadable Modules*** in the System Runtime 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 was 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. See ***Dynamically-Loadable Modules*** in the System Runtime 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. See ***Dynamically-Loadable Modules*** in the System Runtime 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_GetStringTable, 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. See ***Dynamically-Loadable Modules*** in the System Runtime 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.

- `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_GetStringTable, dyn_GetExpSymTab, dyn_LookupByName, dyn_Relocate, dyn_SetSectionAddr, dyn_CopySectionContents, dyn_FreeSectionMem, dyn_AllocSectionMemHeap, malloc

dyn_LookupByName

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 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, fnname, 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_GetStringTable, 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. See ***Dynamically-Loadable Modules*** in the System Runtime 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_RetrieveRelocOutOfRange`.

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` aborts.

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. See ***Dynamically-Loadable Modules*** in the System Runtime 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_RecordRelocOutOfRange 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_GetStringTable, dyn_GetExpSymTab, dyn_LookupByName, dyn_SetSectionAddr, dyn_AllocSectionMem, 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. See *Dynamically-Loadable Modules* in the System Runtime 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` is set to the computed location or value that was being referenced by `*ref_addr`.

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` are 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_RecordRelocOutOfRange

dyn_RewriteImageToFile

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 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. See ***Dynamically-Loadable Modules*** in the System Runtime 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. See *Dynamically-Loadable Modules* in the System Runtime 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. See **Dynamically-Loadable Modules** in the System Runtime 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` bytes 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_GetNumSections, 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 `___lib_prog_term` label, which is defined by this function.

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.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 [-708.39, 709.78]. 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.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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

fclose

Close a stream

Synopsis

```
#include <stdio.h>

int fclose(FILE *stream);
```

Description

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

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

The `fclose` function returns 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, the function returns zero; otherwise it returns 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, `fflush` returns zero.

Error Conditions

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

Example

```
#include <stdio.h>

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

See Also

fclose

fgetc

Get a character from a stream

Synopsis

```
#include <stdio.h>

int fgetc(FILE *stream);
```

Description

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

Upon successful completion, the `fgetc` function returns the next byte from the input stream pointed to by `stream`.

Error Conditions

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

Example

```
#include <stdio.h>

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

See Also

getc

fgetpos

Record the current position in a stream

Synopsis

```
#include <stdio.h>

int fgetpos(FILE *stream, fpos_t *pos);
```

Description

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

Upon successful completion, the `fgetpos` function returns zero.

Error Conditions

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

Example

```
#include <stdio.h>

void aroutine(FILE *fp, char *buffer)
{
    fpos_t pos;
    /* get the current file position */
    if (fgetpos(fp, &pos)!= 0) {
```

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

See Also

fsetpos, ftell, fseek, rewind

fgets

Get a string from a stream

Synopsis

```
#include <stdio.h>

char *fgets(char *s, int n, FILE *stream);
```

Description

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

Upon successful completion, fgets returns s.

Error Conditions

If fgets is unsuccessful, the function returns a NULL pointer.

Example

```
#include <stdio.h>

char buffer[20];
void read_into_buffer(FILE *fp)
{
    char *str;
    str = fgets(buffer, sizeof(buffer), fp);
    if (str == NULL) {
        printf("Either read failed or EOF encountered\n");
```

```
    } else {
        printf("filled buffer with %s\n", str);
    }
}
```

See Also

fgetc, getc, gets

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 invalid or not open. It returns a positive number if successful.

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.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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` function 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` function return zero.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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.
w	Open text file for writing. If the file name already exists, it will be truncated to zero length with the write starting at the beginning of the file. If the file does not already exist, it is created.
a	Open a text file for appending data. All data will be written to the end of the specified file.
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 modify the meaning of the conversion.
- Width - optional numeric value (or `*`) that specifies the minimum field width.
- Precision - optional numeric value that specifies 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 following table.

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 k K r R: 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 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 is 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,k,K,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, 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 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`, `k`, `K`, `r`, `R`, or `n` conversion specifiers unless other conversion specifiers are detailed.

Length	Action
h	The argument should be interpreted as a `shortint`, `short fract`, or `short accum`.
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 `longint`, `long fract`, or `long accum`.
ll	The argument should be interpreted as a `long longint`.
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.
t	The argument should be interpreted as `ptrdiff_t`.
z	The argument should be interpreted as `size_t`.

Note that the `hh`, `j`, `t`, and `z` size specifiers, as described in the C99 (ISO/IEC 9899:1999) standard, are only available if the `-full-io` option has been selected.

The following table contains definitions of the valid conversion specifiers that define the type of conversion to be applied:

Specifier	Conversion
a, A	Floating-point number
c	Character
d, i	Signed decimal integer
e, E	Scientific notation (mantissa/exponent)
f, F	Decimal floating-point
g, G	Convert as e, E or f, F
k	Signed accum
K	Unsigned accum
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.hhhhpd` 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.dddedd`. 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 specifier converts to decimal notation `[-]d.ddd`.

The g|G conversion specifier converts as e|E or f|F specifiers depending on the value being converted. If the exponent of 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 or INF, with the case matching that of the specifier. 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 or NAN, with the case matching that of the specifier.

The k|K and r|R conversion specifiers convert a fixed-point value to decimal notation `[-]d.ddd` when your application is built with the `-full-io` or `-fixed-point-io` switch. Otherwise, the k|K and r|R conversion specifiers convert a fixed-point value to hexadecimal.

The `fprintf` function returns the number of characters printed.

Error Conditions

If the `fprintf` function is unsuccessful, a negative value is returned.

Example

```
#include <stdio.h>

void fprintf_example(void)
{
   char *str = "hello world";
   /* Output to stdout is "  +1  +1." */
   fprintf(stdout, "%+5.0f%+#5.0f\n", 1.234, 1.234);
   /* Output to stdout is "1.234 1.234000 1.23400000" */
   fprintf(stdout, "%.3f %f %.8f\n", 1.234, 1.234, 1.234);
   /* Output to stdout is "justified:
                          left:5    right:    5" */
   fprintf(stdout, "justified:\nleft:%-5dright:%5i\n", 5, 5);
   /* Output to stdout is
      "90% of test programs print hello world" */
   fprintf(stdout, "90%% of test programs print %s\n", str);
   /* Output to stdout is "0.0001 1e-05 100000 1E+06" */
   fprintf(stdout, "%g %g %G %G\n", 0.0001, 0.00001, 1e5, 1e6);
}
```

See Also

printf, snprintf, vfprintf, vprintf, vsnprintf, vsprintf

fputc

Put a character on a stream

Synopsis

```
#include <stdio.h>

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

Description

The `fputs` 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 `fputs` function is successful then it returns the value that was written to the stream.

Error Conditions

If the `fputs` function is not successful, `EOF` is returned.

Example

```c
#include <stdio.h>

void fputc_example(FILE* fp)
{
   /* put the character 'i' to the stream pointed to by fp */
   int res = fputc('i', fp);
   if (res != 'i')
      printf("fputc failed\n");
}
```

See Also

putc

fputs

Put a string on a stream

Synopsis

```c
#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 returns `EOF` if a write error occurred.

Example

```c
#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 an item of data is a sequence of bytes of length `size.` It stops reading bytes 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 bytes read. It does not change the contents of `stream`.

The `fread` function returns the number of items read. This may be less than `n` if there is insufficient data on the external device to satisfy the read request. If `size` or `n` is zero, then `fread` returns zero and does not affect the state of `stream`.

When the stream has been opened as a binary stream, the Analog Devices I/O library may choose to bypass the I/O buffer and transmit data from an external device directly into the program, particularly when the buffer size (as defined by the macro `BUFSIZ` in the `stdio.h` header file or controlled by the function `setvbuf`) is smaller than the number of characters to be transferred. If an application relies on this function to always read data via an I/O buffer, then it should be linked against the third-party library (using the `-full-io` switch).

Error Conditions

If an error occurs, `fread` returns zero and set the error indicator for `stream`.

Example

```
#include <stdio.h>

int buffer[100];
int fill_buffer(FILE *fp)
{
   int read_items;
   /* Read from file pointer fp into array buffer */
```

```
    read_items = fread(&buffer, sizeof(int), 100, fp);
    if (read_items < 100) {
        if (ferror(fp))
            printf("fill_buffer failed with an I/O error\n");
        else if (feof(fp))
            printf("fill_buffer failed with EOF\n");
        else
            printf("fill_buffer only read %d items\n",read_items);
    }
    return read_items;
}
```

See Also

ferror, fgetc, fgets, fscanf

free

Deallocate memory

Synopsis

```
#include <stdlib.h>

void free(void *ptr);
```

Description

The free function deallocates a pointer previously allocated to a range of memory (by calloc or malloc) to the free memory heap. If the pointer was not previously allocated by calloc, malloc, or 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 = (char *)malloc(10);   /* Allocate 10 bytes from heap */
free(ptr);                  /* Return space to free heap   */
```

See Also

calloc, malloc, realloc

freopen

Open a file using an existing 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, `freopen` 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 f, int *expptr);
double frexp(double f, int *expptr);
long double frexpd (long double f, int *expptr);
```

Description

The `frexp` functions separate a floating-point input into a normalized fraction and a (base 2) exponent. The functions return the first argument as a fraction which is 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 the fraction and exponent are both set to zero.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 being pointers to the locations where the converted results are to be written to.

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. The `fscanf` function 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 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`, `k`, `K`, `r`, `R` or `n` conversion specifiers unless other conversion specifiers are detailed.

Length	Action
h	The argument should be interpreted as a `short int`, `short fract`, or `short accum`.
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`, `long fract`, or `long accum`.
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.
t	The argument should be interpreted as `ptrdiff_t`.
z	The argument should be interpreted as `size_t`.

Note that the `hh`, `j`, `t`, and `z` size specifiers are defined in the C99 (ISO/IEC 9899:1999) standard.

A conversion specification terminates with a conversion specifier that defines how the input data is to be converted. The valid conversion specifiers can be found in the following table.

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
k	Signed accum with optional sign
K	Unsigned accum
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
[	Non-empty sequence of characters referred to as the scanset
%	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, no data shall be placed into the object with the next conversion using the object to store its result.

Note that the k, K, r and R format specifiers are only supported when building with either the -full-io or -fixed-point-io switches.

The fscanf function returns the number of items successfully read.

Error Conditions

If the `fscanf` function is not successful before any conversion, `EOF` is returned.

Example

```
#include <stdio.h>

void fscanf_example(FILE *fp)
{
   short int day, month, year;
   float f1, f2, f3;
   char string[20];
   /* Scan a date with any separator, eg, 1-1-2006 or 1/1/2006 */
   fscanf (fp, "%hd%*c%hd%*c%hd", &day, &month, &year);
   /* Scan float values separated by "abc", for example
      1.234e+6abc1.234abc235.06abc                      */
   fscanf (fp, "%fabc%gabc%eabc", &f1, &f2, &f3);
   /* For input "alphabet", string contains "a" */
   fscanf (fp, "%[aeiou]", string);
   /* For input "drying", string contains "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` bytes from the beginning of `stream`.

whence	Effect
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.

NOTE: Positioning within a file that has been opened as a text stream is only supported by the libraries supplied by Analog Devices 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 undo 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.

The fseek function returns zero when successful.

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 bytes offset from the start of fp */
   if (fseek(fp, 20, SEEK_SET) != 0) {
      printf("fseek failed\n");
      return -1;
   }
   /* Now use ftell to get the offset value back */
   offset = ftell(fp);
   if (offset == -1)
      printf("ftell failed\n");
   if (offset == 20)
      printf("ftell and fseek work\n");
   return offset;
}
```

See Also

fflush, ftell, ungetc

fsetpos

Reposition a file pointer in a stream

Synopsis

```
#include <stdio.h>

int fsetpos(FILE *stream, const fpos_t *pos);
```

Description

The fsetpos function sets the file position indicator for stream, using the value of the object pointed to by pos. The value pointed to by pos must be a value obtained from an earlier call to fgetpos on the same stream.

> **NOTE:** Positioning within a file that has been opened as a text stream is only supported by the libraries supplied by Analog Devices if the lines within the file are terminated by the character sequence \r\n.

A successful call to fsetpos clears the EOF indicator for stream and undoes any effects of ungetc on the same stream.

The fsetpos function returns zero if it is successful.

Error Conditions

If the fsetpos function is unsuccessful, the function returns a non-zero value.

Example

Refer to fgetpos for an example.

See Also

fgetpos, fseek, 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 binary stream, then the value is the number of characters from the beginning of the file. If `stream` is a text stream, then the information in the position indicator is unspecified information which is usable by `fseek` for determining the file position indicator at the time of the `ftell` call.

> **NOTE:** Positioning within a file that has been opened as a text stream is only supported by the libraries supplied by Analog Devices if the lines within the file are terminated by the character sequence \r\n.

If successful, `ftell` 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

Refer to 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 an application relies on this feature to always write data via an I/O buffer, then it should be linked against the third-party I/O library, using the `-full-io` switch.

If successful, the `fwrite` function returns the number of items written.

Error Conditions

If the `fwrite` function is unsuccessful, it returns the number of elements successfully written, which is less than `n`.

Example

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

char* message="some text";

void write_text_to_file(void)
{
   /* Open "file.txt" for writing */
   FILE* fp = fopen("file.txt", "w");
   int res, message_len = strlen(message);
   if (!fp) {
      printf("fopen was not successful\n");
      return;
   }
   res = fwrite(message, sizeof(char), message_len, fp);
   if (res != message_len)
      printf("fwrite was not successful\n");
}
```

See Also

fread

fxbits

Bitwise integer to fixed-point to conversion

Synopsis

```
#include <stdfix.h>

short fract hrbits(int_hr_t b);
fract rbits(int_r_t b);
long fract lrbits(int_lr_t b);
unsigned short fract uhrbits(uint_uhr_t b);
unsigned fract urbits(uint_ur_t b);
unsigned long fract ulrbits(uint_ulr_t b);
short accum hkbits(int_hk_t b);
accum kbits(int_k_t b);
long accum lkbits(int_lk_t b);
unsigned short accum uhkbits(uint_uhk_t b);
unsigned accum ukbits(uint_uk_t b);
unsigned long accum ulkbits(uint_ulk_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>

accum k;
unsigned long fract ulr;
k = kbits(-0x6400000000ll);           /* k == -12.5k */
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);
accum kdivi(int numer, int denom);
long accum lkdivi(long int numer, long int denom);
unsigned accum ukdivi(unsigned int numer, unsigned int denom);
unsigned long accum ulkdivi(unsigned long int numer,
                            unsigned long int denom);
```

Description

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

Error Conditions

The `fxdivi` family of functions have undefined behavior if the denominator is zero.

Example

```
#include <stdfix.h>

accum quo;
unsigned long fract ulquo;
quo = kdivi(125, -10);          /* quo == -12.5k */
ulquo = ulrdivi(1, 8);          /* ulquo == 0.125ulr */
```

See Also

divifx, idivfx

getc

Get a character from a stream

Synopsis

```
#include <stdio.h>

int getc(FILE *stream);
```

Description

The `getc` function is functionally equivalent to `fgetc`, except that it is implemented (if `-full-io` is specified) as a macro for C language dialects and as an inline function if the language dialect is C++.

The resulting implementation will be more efficient than making a call to the `fgetc` function, though there are considerations on code size and the inability to pass the address of `getc` to another function.

Note that if the `-no-full-io` switch option is specified, then `getc` is implemented as a standard function call.

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` returns the next single character from the standard input stream. The `getchar` function also advances the standard input's current position indicator.

The `getchar` function is implemented (if the `-full-io` switch option is specified) as a macro for C language dialects and as an inline function if the language dialect is C++.

The resulting implementation is more efficient than making a function call, though there are considerations on code size and the ability to pass the address of `getchar` to another function.

Note that if the `-no-full-io` is specified, then `getchar` is implemented as a standard function call.

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

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 will terminate 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, 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, as 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 `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 from a heap

Synopsis

```
#include <stdlib.h>

void *heap_calloc(int heap_index, size_t nelem, size_t size);
```

Description

The `heap_calloc` function allocates an array from the heap identified by `heap_index`. The array contains `nelem` elements, each of size `size`; the whole array will be initialized to zero.

The function returns a pointer to the array. The return value can be safely converted to an object of any type whose size is not greater than `size*nelem` bytes. The memory allocated by `calloc` may be deallocated by either the `free` or `heap_free` functions.

Note that the `userid` of a heap is not the same as the heap's index; the index of a heap is returned by the function `heap_install` or `heap_lookup`. Refer to Using the Alternate Heap Interface for more information on multiple run-time heaps.

Error Conditions

The `heap_calloc` function returns a null pointer if the requested memory could not be allocated.

Example

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

int heapid = HEAP1_USERID;
int heapindex = -1;
long *alloc_array(int nels)
{
   if (heapindex < 0) {
      heapindex = heap_lookup(heapid);
      if (heapindex == -1) {
         printf("Heap %d is not defined\n",heapid);
         exit(EXIT_FAILURE);
      }
   }
   return heap_calloc(heapindex,nels,sizeof(long));
}
```

See Also

calloc, free, heap_free, heap_init, heap_install, heap_lookup, heap_malloc, heap_realloc, heap_space_unused, malloc, realloc, 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 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 is allocated and returns the memory to this heap. For more information on creating multiple run-time heaps, refer to Using Multiple Heaps.

Error Conditions

None

Example

```
#include <stdlib.h>

extern int userid;
int heapindex = heap_lookup(userid);
char *ptr = heap_malloc(heapindex,32 * sizeof(char));
    ...
heap_free(0,ptr);
```

See Also

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

heap_init

Re-initialize a heap

Synopsis

```
#include <stdlib.h>

int heap_init(int heap_index);
```

Description

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_init, heap_install, heap_lookup, heap_malloc, heap_realloc, heap_space_unused, malloc, realloc, space_unused

heap_install

Set up a heap at runtime

Synopsis

```
#include <stdlib.h>

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

Description

The `heap_install` function initializes the heap identified by the parameter `userid`. The heap will be set up at the address specified by `base` and with a size in bytes specified by `length`. The function returns the heap index for the heap once it has been successfully initialized.

Not all length bytes will be available for allocation from the heap, as some space is claimed for administration, and some space is required, per allocation. For more information, see Tips for Working With Heaps.

The function `heap_malloc` and the associated functions, such as `heap_calloc` and `heap_realloc`, may be used to allocate memory from the heap once the heap has been initialized. Refer to Using Multiple Heaps for more information.

To re-initialize a heap that is already installed, use the heap_init function.

Error Conditions

The `heap_install` function returns -1 if the heap was not initialized successfully. 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; the space wraps around the end of the address space.

Example

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

static int heapid = 0;
int setup_heap(void *at, size_t bytes)
{
    int index;
    if ( (index = heap_install(at, bytes, ++heapid)) == -1) {
        printf("Failed to initialize heap with userid %d\n",
                                                  heapid);
        exit(EXIT_FAILURE);
    }
    return index;
}
```

See Also

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

heap_lookup

Convert a `userid` to a heap index

Synopsis

```
#include <stdlib.h>

int heap_lookup(int userid);
```

Description

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, refer to Using Multiple Heaps in the *Compiler* chapter.

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("Heap %d not setup
          -- will use the default heap\n", heap_userid);
   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_free, heap_init, heap_install, heap_lookup, heap_malloc, heap_realloc, heap_space_unused, 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 allocates an object of `size` bytes, from the heap with heap index `heap_index`. It returns the address of the object if successful. The return value may be used as a pointer to an object of any type whose size in bytes is not greater than `size`.

The block of memory returned is uninitialized. The memory may be deallocated with either the `free` or `heap_free` function. For more information on creating multiple run-time heaps, refer to Using Multiple Heaps in the *Compiler* chapter.

Error Conditions

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

Example

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

int heap_index = heap_lookup(USERID_HEAP);
long *buffer;
if (heap_index < 0) {
   printf("Heap %d is not setup\n",USERID_HEAP);
   exit(EXIT_FAILURE);
}
buffer = heap_malloc(heap_index,16 * sizeof(long));
if (buffer == NULL) {
   printf("heap_malloc failed to allocate memory\n");
}
```

See Also

calloc, free, heap_free, heap_init, heap_install, heap_lookup, heap_malloc, heap_realloc, heap_space_unused, malloc, realloc, 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 changes the size of a previously allocated block of memory. The new size of the object in bytes is specified by the argument size; the new object retains the values of the old object up to its original size, but any data beyond the original size will be indeterminate. The address of the object is given by the argument ptr. The behavior of the function is not defined if either the object has not been allocated from a heap, or if it has already been freed.

If ptr is a null pointer, then heap_realloc behaves the same as heap_malloc. 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.

If the function successfully re-allocates the object, then it returns a pointer to the new object.

Error Conditions

If heap_realloc cannot reallocate the memory, it returns a null pointer and the original memory associated with ptr will be unchanged and will still be available.

Example

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

int heap_index = heap_lookup(USERID_HEAP);
int *buffer;
int *temp_buffer;

   if (heap_index < 0) {
   printf("Heap %d is not setup\n",USERID_HEAP);
   exit(EXIT_FAILURE);
}
buffer = heap_malloc(heap_index,32*sizeof(int));
if (buffer == NULL) {
   printf("heap_malloc failed to allocate memory\n");
}
...
temp_buffer = heap_realloc(0,buffer,64*sizeof(int));
if (temp_buffer == NULL) {
   printf("heap_realloc failed to allocate memory\n");
} else {
   buffer = temp_buffer;
}
```

See Also

calloc, free, heap_free, heap_init, heap_install, heap_lookup, heap_malloc, heap_realloc, heap_space_unused, malloc, realloc, 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 returns the total free space in bytes 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;
free_space = heap_space_unused(1);  /* Get free space in heap 1 */
```

See Also

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

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);
int idivk(accum numer, accum denom);
long int idivlk(long accum numer, long accum denom);
```

```
unsigned int idivuk(unsigned accum numer, unsigned accum denom);
unsigned long int idivulk(unsigned long accum numer,
                          unsigned long accum 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` family of functions have undefined behavior if the denominator is zero.

Example

```
#include <stdfix.h>

int quo;
unsigned long int ulquo;

quo = idivk(125.0k, -12.5k);              /* quo == -10 */
ulquo = idivulr(0.5ulr, 0.125ulr);        /* ulquo == 4 */
```

See Also

divifx, fxdivi

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 will occur 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 which is not built with instrumented profiling

enabled (see the -p switch). You can guard such code using the preprocessor macro _INSTRUMENTED_PROFILING; 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 *Achieving Optimal Performance From C/C++ Source Code* chapter.

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)
    #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 whether the argument is an alphanumeric character (`A-Z`, `a-z`, or `0-9`). If the argument is not alphanumeric, `isalnum` returns a zero. If the argument is alphanumeric, `isalnum` returns a non-zero value.

The function's behavior is only defined if the argument `c` is either `EOF`, or is equivalent to an `unsigned char`.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 whether the input is an alphabetic character (A-Z or a-z). If the input is not alphabetic, isalpha returns a zero. If the input is alphabetic, isalpha returns a non-zero value.

The function's behavior is only defined if the argument c is either EOF, or is equivalent to an unsigned char.

NOTE: This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 whether 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.

The function's behavior is only defined if the argument `c` is either `EOF`, or is equivalent to an `unsigned char`.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 whether the input character is a decimal digit (0-9). If the input is not a digit, `isdigit` returns a zero. If the input is a digit, `isdigit` returns a non-zero value.

The function's behavior is only defined if the argument `c` is either `EOF`, or is equivalent to an `unsigned char`.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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, isxdigit

isgraph

Detect printable character, not including white space

Synopsis

```
#include <ctype.h>

int isgraph(int c);
```

Description

The `isgraph` function determines whether the argument is a printable character, not including 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.

The function's behavior is only defined if the argument `c` is either `EOF`, or is equivalent to an `unsigned char`.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 isinf(double x);
int isinff(float x);
int isinfd (long double x);
```

Description

The isinf functions return a zero if the argument is not set to the IEEE constant for +Infinity or –Infinity; otherwise, the functions return a non-zero value.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

None

Example

```
#include <math.h>

static int fail=0;
```

```
main(){
   /* test int isinf(double) */
   union {
           double d; float f; unsigned long l;
   } u;

   #ifdef __DOUBLES_ARE_FLOATS__
      u.l=0xFF800000L; if ( isinf(u.d)==0 ) fail++;
      u.l=0xFF800001L; if ( isinf(u.d)!=0 ) fail++;
      u.l=0x7F800000L; if ( isinf(u.d)==0 ) fail++;
      u.l=0x7F800001L; if ( isinf(u.d)!=0 ) fail++;
   #endif

   /* test int isinff(float) */
      u.l=0xFF800000L; if ( isinff(u.f)==0 ) fail++;
      u.l=0xFF800001L; if ( isinff(u.f)!=0 ) fail++;
      u.l=0x7F800000L; if ( isinff(u.f)==0 ) fail++;
      u.l=0x7F800001L; if ( isinff(u.f)!=0 ) fail++;

   /* print pass/fail message */
   if ( fail==0 )
      printf("Test passed\n");
   else
      printf("Test failed: %d\n", fail);
}
```

See Also

isnan

islower

Detect lowercase character

Synopsis

```
#include <ctype.h>

int islower(int c);
```

Description

The islower function determines whether 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.

The function's behavior is only defined if the argument c is either EOF, or is equivalent to an unsigned char.

NOTE: This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 return a zero if the argument is not set to an IEEE NaN; otherwise, the functions return a non-zero value.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

None

Example

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

static int fail=0;
```

```
main(){
   /* test int isnan(double) */
   union {
      double d; float f; unsigned long l;
   } u;
   #ifdef __DOUBLES_ARE_FLOATS__
      u.l=0xFF800000L; if ( isnan(u.d)!=0 ) fail++;
      u.l=0xFF800001L; if ( isnan(u.d)==0 ) fail++;
      u.l=0x7F800000L; if ( isnan(u.d)!=0 ) fail++;
      u.l=0x7F800001L; if ( isnan(u.d)==0 ) fail++;
   #endif
   /* test int isnanf(float) */
      u.l=0xFF800000L; if ( isnanf(u.f)!=0 ) fail++;
      u.l=0xFF800001L; if ( isnanf(u.f)==0 ) fail++;
      u.l=0x7F800000L; if ( isnanf(u.f)!=0 ) fail++;
      u.l=0x7F800001L; if ( isnanf(u.f)==0 ) fail++;
   /* print pass/fail message */
   if ( fail==0 )
      printf("Test passed\n");
   else
      printf("Test failed: %d\n", fail);
}
```

See Also

isinf

isprint

Detect printable character

Synopsis

```
#include <ctype.h>

int isprint(int c);
```

Description

The isprint function determines whether 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.

The function's behavior is only defined if the argument c is either EOF, or is equivalent to an unsigned char.

NOTE: This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 whether 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.

The function's behavior is only defined if the argument `c` is either `EOF`, or is equivalent to an `unsigned char`.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 whether the argument is a blank whitespace character (0x09-0x0D or 0x20). This includes the characters space ( ), form feed (\f), new line (\n), carriage return (\r), horizontal tab (\t), and vertical tab (\v).

If the argument is not a blank space character, isspace returns a zero. If the argument is a blank space character, isspace returns a non-zero value.

The function's behavior is only defined if the argument c is either EOF, or is equivalent to an unsigned char.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 whether 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.

The function's behavior is only defined if the argument c is either EOF, or is equivalent to an unsigned char.

NOTE: This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 whether 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.

The function's behavior is only defined if the argument `c` is either `EOF`, or is equivalent to an `unsigned char`.

NOTE: This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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

_l1_memcpy, _memcpy_l1

Copy instructions between L1 instruction memory and data memory

Synopsis

```
#include <ccblkfn.h>

void *_l1_memcpy(void *datap, const void *instrp, size_t n);
void *_memcpy_l1(void *instrp, const void *datap, size_t n);
```

Description

The _l1_memcpy function copies n bytes of program instructions from the address `instrp` to the data buffer `datap`. The _memcpy_l1 function is the inverse: it copies n characters of program instructions from the data buffer `datap` to the address `instrp`. For Blackfin parts the following restrictions apply to both functions:

- `n` must be a multiple of 8

- `instrp` must be an address in L1 instruction memory

- `instrp` must be 8-byte aligned

- `datap` must be 4-byte aligned

- `instrp+n-1` must be within L1 instruction memory

For Blackfin+ parts, which support Extended Access Mode, no error checking is carried out and both functions behave as a regular memcpy with Extended Access Mode enabled.

Note that for the ADSP-BF6xx processors, these functions invalidate the cache. If parity checking is enabled, parity within the cache will also be invalidated, and so will need re-initializing. The functions only perform parity re-initialization if parity checking was enabled, in order to reduce overhead. If you are calling these functions several times successively, you may consider disabling parity checking for the duration of the sequence of calls, as this will avoid the overhead of re-initializing parity data that is immediately invalidated by the following call.

The _l1_memcpy function returns `datap` for success. The _memcpy_l1 function returns `instrp` for success.

The C and C++ run-time libraries use _memcpy_l1 to implement the memory-initialization process, if the `.dxe` file has been built with the `-mem` compiler switch, or with the `-meminit` linker switch.

Error Conditions

If any of the restrictions are not met, the _l1_memcpy and _memcpy_l1 functions return `NULL`.

NOTE: On platforms where L1_CODE_CACHE does not follow on directly from L1_CODE in memory (such as ADSP-BF561, ADSP-BF52x, ADSP-BF531, ADSP-BF534, ADSP-BF536, ADSP-BF537, and ADSP-BF54x processors), _l1_memcpy and _memcpy_l1 allow users to write to any memory in between. Ensure that addresses being written to are entirely within valid L1_CODE or L1_CODE_CACHE.

Example

```
/* copying program instructions from L1 Instruction
** memory to data memory.
*/
#include <ccblkfn.h>

char dest[32];
const char *src = (const char *) /* address in L1 code space */;
if (_l1_memcpy(dest, src, 32) != dest)
```

```
   exit(1);

/* copying program instructions from data memory
** to L1 Instruction memory.
*/
#include <ccblkfn.h>

const char src[32] = { /* some instruction op-codes */ };
char *dest = (char *) /* Address in L1 code space */;
if (_memcpy_l1(dest, src, 32) != dest)
   exit(1);
```

See Also

[memcpy](#)

labs

Long integer absolute value

Synopsis

```
#include <stdlib.h>

long int labs(long int j);
long long int llabs (long long int j);
```

Description

The `labs` and `llabs` functions return the absolute value of their integer inputs.

> **NOTE:** The result of `labs(LONG_MIN)` is undefined.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See [ADSP-BF7xx Processors L2 Utility ROM](#) for further information.

Error Conditions

None

Example

```
#include <stdlib.h>

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

See Also

abs, absfx, fabs

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.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

If the result overflows, the ldexp functions return HUGE_VAL with the proper sign. If the result underflows, the functions return a zero. In addition, ldexpf (and ldexp if the size of the double type is the same as the size of the float type) will set errno to ERANGE.

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);
lldiv_t lldiv (long long int numer, long long int denom);
```

Description

The `ldiv` and `lldiv` functions divide `numer` by `denom` and return a structure of type `ldiv_t` and `lldiv_t`, respectively. The types `ldiv_t` and `lldiv_t` are defined as:

```
typedef struct {
    long int quot;
    long int rem;
} ldiv_t;

typedef struct {
    long long int quot;
    long long int rem;
} lldiv_t;
```

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

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

If `denom` is zero, the behavior of the `ldiv` and `lldiv` functions are undefined.

Example

```
#include <stdlib.h>

ldiv_t result;
result = ldiv(7, 2);        /* result.quot=3, result.rem=1 */
```

See Also

div, divifx, fmod, fxdivi, idivfx

localtime

Convert calendar time into broken-down time

Synopsis

```
#include <time.h>

struct tm *localtime(const time_t *t);
```

Description

The `localtime` function converts a pointer to a calendar time into a broken-down time that corresponds to current time zone. A broken-down time is a structured variable, which is described in time.h. This implementation of the header file does not support the Daylight Saving flag nor does it support time zones and, thus, `localtime` is equivalent to the `gmtime` function.

The broken-down time is returned by `localtime` as a pointer to static memory, which may be overwritten by a subsequent call to either `localtime` or to `gmtime`.

Error Conditions

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.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The natural logarithm functions return −HUGE_VAL if the input value is zero or negative.

Example

```
#include <math.h>

double y;
float x;

y = log (1.0);          /* y = 0.0 */
x = logf (2.71828);     /* x = 1.0 */
```

See Also

alog, exp, log10

log10

Base 10 logarithm

Synopsis

```
#include <math.h>

float log10f (float f);
double log10(double f);
long double log10d (long double f);
```

Description

The log10 functions return the base 10 logarithm of their inputs.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The log10 functions return −HUGE_VAL 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

alog10, 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.

CAUTION: 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. 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 setjump_example(void)
{
   if ((res = setjmp(env)) != 0) {
      printf ("Problem %d reported by func ()", 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 not initialized. The memory allocated is aligned to an 8-byte boundary.

Error Conditions

The malloc function returns a null pointer if it is unable to allocate the requested memory.

Example

```
#include <stdlib.h>

long *ptr;
ptr = (long *)malloc(10 * sizeof(long));  /* ptr points to an */
                                          /* array of 10 longs */
```

See Also

calloc, realloc, free

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.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

None

Example

```
#include <string.h>

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

See Also

strchr, strrchr

memcmp

Compare objects

Synopsis

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);
```

Description

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

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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, strncmp

memcpy

Copy characters from one object to another

Synopsis

```
#include <string.h>

void *memcpy(void *s1, const void *s2, size_t n);
```

Description

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

NOTE: The compiler will always align vectors and arrays on a 32-bit word boundary, and the compiler will normally use this knowledge to replace a call to `memcpy` by more efficient in-line code. The alignment assumptions made by the compiler are safe, provided that the vectors and arrays were allocated by the compiler. If the vectors and arrays were allocated via an assembly function, that assembly function must ensure that the objects `s1` and `s2` are aligned on a 4-byte address boundary; this is normally achieved by preceding the definition of `s1` and `s2` with the `.align 4` assembly directive.

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

NOTE: This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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`.

NOTE: This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

None

Example

```
#include <string.h>

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

See Also

memmove, 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.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

None

Example

```
#include <string.h>

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

See Also

memcpy

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.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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);
int mulik(int a, accum b);
long int mulilk(long int a, long accum b);
unsigned int muliuk(unsigned int a, unsigned accum b);
unsigned long int muliulk(unsigned long int a,
                          unsigned long accum 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>

int prod;
unsigned long int ulprod;
```

```
prod = mulik(128, -1.25k);               /* prod == -160 */
ulprod = muliulr(128, 0.125ulr);         /* ulquo == 16 */
```

See Also

No related functions

perror

Print an error message on standard error

Synopsis

```
#include <stdio.h>

int perror(const char *s);
```

Description

The perror function is used to output an error message to the standard stream stderr.

If string s is not a null pointer and the first character addressed by s is not a null character, the function outputs 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 <stdlib.h>
#include <errno.h>

#define BASE_10 10

int n;

n = strtol ("987654321",NULL,BASE_10);
if (errno != 0)
    perror ("strtol failed");
```

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 run-time support for profile-guided optimization on hardware should write gathered data to the host computer. The flush will occur 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 that is not built for profile-guided optimization on hardware (see the -pguide and -prof-hw switches in the *Compiler* chapter). 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.

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 *Optimal Performance from C/C++ Source Code* chapter.

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.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The power functions return zero when the first argument x is zero and the second argument y is not an integral value. When x is zero and y is less than zero, or when the result cannot be represented, the functions return the constant HUGE_VAL.

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 `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 `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 returns `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)`.

The function is implemented as an inline function if language dialect is C++; for other C language dialects, it is implemented as a macro if switch `-full-io` is specified. When it is implemented as a macro, the resulting implementation is more efficient than making a function call, though there are considerations on code size and the ability to pass the address of `putchar` to another function.

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 returns `EOF`.

Example

```
#include <stdio.h>

void putchar_example(void)
{
   /* write the character 'a' to stdout */
   if (putchar('a') == EOF)
```

```
        fprintf(stderr, "putchar failed\n");
}
```

See Also

putc

puts

Put a string to `stdout`

Synopsis

```
#include <stdio.h>

int puts(const char *s);
```

Description

The `puts` function writes the string pointed to by `s`, followed by a `NEWLINE` character, to the standard output stream `stdout`. The terminating null character of the string is not written to the stream.

If the function call is successful, then the return value is zero or greater.

Error Conditions

The macro `EOF` is returned if `puts` was unsuccessful, and the error indicator for `stdout` will be set.

Example

```
#include <stdio.h>

void puts_example(void)
{
   /* write the string "example" to stdout */
   if (puts("example") < 0)
      fprintf(stderr, "puts failed\n");
}
```

See Also

fputs

qsort

Quicksort

Synopsis

```
#include <stdlib.h>

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

Description

The `qsort` function sorts an array of `nelem` objects, pointed to by `base`. Each object is specified by its `size`.

The contents of the array are sorted into ascending order according to a comparison function pointed to by `compare`, which is called with two arguments that point to the objects being compared. The function returns 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. Note that:

- `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

- `compare` is a pointer to a function that is called by `qsort` to compare two elements of the array. The function returns 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 Condition

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*.

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 $[0, 2^{30}- 1]$.

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 $2^{30}- 1$.

Error Conditions

None

Example

```
#include <stdlib.h>

int i;
i = rand();
```

See Also

srand

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 the values 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. The memory allocated is aligned to a 8-byte boundary.

If `ptr` is a non-null pointer that was not allocated with `malloc` or `calloc`, the behavior is undefined. If `ptr` is a null pointer, `realloc` imitates `malloc`. If `size` is zero and `ptr` is not a null pointer, `realloc` imitates `free`.

Error Conditions

If memory cannot be allocated, `ptr` remains unchanged and `realloc` returns a null pointer.

Example

```
#include <stdlib.h>
int *ptr;

ptr = malloc(10 * sizeof(int));        /* ptr points to an array
                                          of 10 ints          */
ptr = realloc(ptr,20 * sizeof(int));  /* ptr now points to an
                                          array of 20 ints    */
```

See Also

calloc, free, 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 establishes a new name, using the string `newname`, for a file currently known by the string `oldname`. After being successful renamed, the file is no longer 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);
short accum roundhk(short accum a, int n);
accum roundk(accum a, int n);
long accum roundlk(long accum a, int n);
unsigned short accum rounduhk(unsigned short accum a, int n);
unsigned accum rounduk(unsigned accum a, int n);
unsigned long accum roundulk(unsigned long accum a, 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 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>

accum a;
long fract f;
a = roundhk(-12.51k, 1);         /* a == 12.5k */
a = roundfx(-12.51k, 1);         /* a == 12.5k */
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.

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 the number of successful conversions performed.

Error Conditions

The scanf function returns 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 file or 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.

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

setvbuf

setjmp

Define a run-time label

Synopsis

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

Description

The `setjmp` function saves the calling environment in the `jmp_buf` argument. The effect of the call is to declare a run-time label that can be jumped to via a subsequent call to `longjmp`.

When `setjmp` is called, it immediately returns with a result of zero to indicate that the environment has been saved in the `jmp_buf` argument. If, at some later point, `longjmp` is called with the same `jmp_buf` argument, `longjmp` restores the environment from the argument. The execution then resumes 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.

CAUTION: 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. 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

setvbuf

Specify buffering for a file or 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. If buf is non-NULL, 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.

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 returns a non-zero value if either an invalid value is given for type, 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 4-46: 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 *System Run-Time Documentation*.

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 4-47: 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>

double sin (double x);
float sinf (float x);
long double sind (long double x);
fract16 sin_fr16 (fract16 x);
fract32 sin_fr32 (fract32 x);
_Fract sin_fx16 (_Fract x);
long _Fract sin_fx32 (long _Fract x);
```

Description

The sine functions return the sine of the argument. Both the argument x and the results returned by the functions are in radians.

sin_fr16, sin_fr32, sin_fx16 and sin_fx32 sin functions input a fractional value in the range [-1.0, 1.0) corresponding to [-π/2, π/2]. The domain represents half a cycle which can be used to derive a full cycle if required. (See *Notes* below.) The result, in radians, is in the range [-1.0, 1.0).

The domain of sinf is [-102940.0, 102940.0], and the domain for sind is [-843314852.0, 843314852.0]. The result returned by the functions sin, sinf, and sind is in the range [-1, 1]. The functions return 0.0 if the input argument x is outside the respective domains.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

None

Example

```
#include <math.h>

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

Notes

The domain of the sin_fr16, sin_fr32, sin_fx16 and sin_fx32 functions is restricted to the fractional range [-1, 1), which corresponds to half a period from -(π/2) to π/2. It is possible to derive the full period using the following properties of the function.

```
sine [0, π/2] = -sine [π, 3/2 π]

sine [-π/2, 0] = -sine [π/2, π]
```

The function below uses these properties to calculate the full period (from 0 to 2π) of the sine function using an input domain of [0, 0x7fff].

```
#include <math.h>

fract16 sin2pi_fr16 (fract16 x)
{
   if (x < 0x2000) {                        /* <0.25  */
      /* first quadrant [0..π/2):            */
      /* sin_fr16([0x0..0x7fff]) = [0..0x7fff)   */
      return sin_fr16(x * 4);
   } else if (x == 0x2000) {        /* = 0.25  */
      return 0x7fff;
   } else if (x < 0x6000) {         /* < 0.75  */
      /* if (x < 0x4000)                     */
      /* second quadrant [π/2..π):           */
      /* -sin_fr16([0x8000..0x0)) = [0x7fff..0)  */
      /*                                     */
      /* if (x < 0x6000)                     */
      /* third quadrant [π..3/2π):           */
      /* -sin_fr16([0x0..0x7fff]) = [0..0x8000)  */
      return -sin_fr16((0xc000 + x) * 4);
   } else {
      /* fourth quadrant [3/2π..π):          */
      /* sin_fr16([0x8000..0x0)) = [0x8000..0)   */
      return sin_fr16((0x8000 + x) * 4);
   }
}
```

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`.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The input argument `x` must be in the domain [-87.33, 88.72] for `sinhf`, and in the domain [-710.46, 710.47] for `sinhd`. If the input value is greater than the function's domain, `HUGE_VAL` is returned; if the input value is less than the domain, `-HUGE_VAL` is returned.

Example

```
#include <math.h>

double x, y;
float z, w;

y = sinh (x);
z = sinhf (w);
```

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 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 NUL character is written after the end of the last

character written to the output array unless n is set to zero, in which case nothing will be written to the output array and the output array may be represented by the NULL pointer.

The 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 contains 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 total free space in bytes 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 returns -1.

Example

```
#include <stdlib.h>

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

See Also

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

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 returns 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);
fract16 sqrt_fr16 (fract16 x);
fract32 sqrt_fr32 (fract32 x);
_Fract sqrt_fx16 (_Fract x);
long _Fract sqrt_fx32 (long _Fract x);
```

Description

The square root functions return the positive square root of the argument x.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

If the input argument is negative, then functions `sqrtf`, `sqrt` and `sqrtd` return a `NaN`, while the functions `sqrt_fr16`, `sqrt_fr32`, `sqrt_fx16` and `sqrt_fx32` return a zero.

Example

```
#include <math.h>

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

srand

Random number seed

Synopsis

```
#include <stdlib.h>

void srand(unsigned int seed);
```

Description

The `srand` function sets 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. The function returns a pointer to the new s1 string, which is null-terminated. The behavior of strcat is undefined if the two strings overlap.

NOTE: This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 (null-terminated string) that contains the character c.

NOTE: This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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, strstr

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. 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.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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

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 `LC_COLLATE` locale macro. Because only the C locale is defined in the run-time environment, the `strcoll` function is identical to the `strcmp` function. The function returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the `s2` string is greater than the `s1` string, and a zero if the strings are the same.

Error Conditions

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

strcmp, strncmp

strcpy

Copy from one string to another

Synopsis

```
#include <string.h>

void *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`. The 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`.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 length of the initial segment of `s1`, which consists entirely of characters not 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.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

None

Example

```
#include <string.h>

char *ptr1, *ptr2;
size_t len;
ptrl = "Tried and Tested";
ptr2 = "aeiou";
len = strcspn (ptrl,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 returns a pointer to a string containing an error message by mapping the number in errnum to that string.

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

Table 4-48: 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
`%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

Table 4-48: Conversion Specifiers Supported by strftime (Continued)

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

NOTE: This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The `strftime` function returns zero if more than `buf_size` characters are required to process the format string. In this case, the contents of the array `buf` will be indeterminate.

Example

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

extern void print_time(time_t tod)
{
   char tod_string[100];
   strftime(tod_string,
            100,
            "It is %M min and %S secs after %l o'clock (%p)",
            gmtime(&tod));
   puts(tod_string);
}
```

See Also

ctime, gmtime, localtime, mktime

strlen

String length

Synopsis

```
#include <string.h>

size_t strlen(const char *s1);
```

Description

The strlen function returns the length of the null-terminated string pointed to by s1 (not including the termi-nating null character).

NOTE: This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utili-ty ROM for further information.

Error Conditions

None

Example

```
#include <string.h>

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

See Also

strcspn , strspn

strncat

Concatenate characters from one string to another

Synopsis

```
#include <string.h>

char *strncat(char *s1, const char *s2, size_t n);
```

Description

The strncat function appends a copy of up to n characters in the null-terminated string pointed to by s2 to the end of the null-terminated string pointed to by s1. The function 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').

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

None

Example

```
#include <string.h>

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

See Also

strcat

strncmp

Compare characters in strings

Synopsis

```
#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n);
```

Description

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

NOTE: This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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, strcmp

strncpy

Copy characters from one string to another

Synopsis

```
#include <string.h>

char *strncpy(char *s1, const char *s2, size_t n);
```

Description

The strncpy function copies up to n characters of the null-terminated string 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 when 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 are written.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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, strcpy

strpbrk

Find character match in two strings

Synopsis

```
#include <string.h>

char *strpbrk(const char *s1, const char *s2);
```

Description

The `strpbrk` function returns a pointer to the first character in `s1` that is also found in `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

In the event that no character in `s1` matches any in `s2`, a null pointer is returned.

Example

```
#include <string.h>

char *ptr1, *ptr2, *ptr3;
ptr1 = "TESTING";
ptr2 = "SHOP";
ptr3 = strpbrk(ptr1, ptr2);
   /* ptr3 points to the S in TESTING */
```

See Also

strspn

strrchr

Find last occurrence of character in string

Synopsis

```
#include <string.h>

char *strrchr(const char *s1, int c);
```

Description

The strrchr function returns a pointer to the last occurrence of character c in the null-terminated input string s1.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 of `s1` of the characters pointed to by `s2`. This excludes the terminating null character in `s1`.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 `isspace`) 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 is 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

strtof

Convert string to float

Synopsis

```
#include <stdlib.h>

float strtof (const char *nptr, char **endptr);
```

Description

The `strtof` function extracts a value from the string pointed to by `nptr`, and returns the value as a `float`. The `strtof` 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 `isspace`) 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 `strtof` function returns a zero if no conversion is 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;
float ff;

ff = strtof ("2345.5E4 abc",&rem);
     /* ff = 2.3455E+7, rem = " abc" */

ff = strtof ("-0x1.800p+9,123",&rem);
     /* ff = -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);
short accum strtofxhk(const char *nptr, char **endptr);
accum strtofxk(const char *nptr, char **endptr);
long accum strtofxlk(const char *nptr, char **endptr);
unsigned short accum strtofxuhk(const char *nptr, char **endptr);
unsigned accum strtofxuk(const char *nptr, char **endptr);
unsigned long accum strtofxulk(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 isspace) 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;
accum k;

unsigned long fract ulr;

k = strtofxk ("-2345.5E-3 abc",&rem);
     /* k = -2.3455k, rem = " abc" */

ulr = strtofxulr ("0x180p-12,123",&rem);
     /* ulr = 0x1800p-16ulr, rem = ",123"   */
```

See Also

strtod, strtol, strtoul

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 the string 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. The s1 string is modified in place to insert a null character at the end of the returned token. 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, and unrecognized characters, including a terminating null character. The initial characters may comprise an optional sign character, `0x` or `0X`, when `base` is 16, and those letters and digits which represent an integer with a radix of `base`. The letters (`a-z` or `A-Z`) are assigned the values 10 to 35 and are permitted only when those values are less than the value of `base`.

If `base` is zero, 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 is 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. The `ERANGE` value 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, strtoul

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 `isspace`) 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

strtofxfx, strtol, strtoul

strtoll

Convert string to long long integer

Synopsis

```
#include <stdlib.h>

long long int strtoll(const char *nptr, char **endptr, int base);
```

Description

The strtoll function returns as a long long int the value represented by the string nptr. If endptr is not a null pointer, strtoll stores a pointer to the unconverted remainder in *endptr.

The strtoll function breaks down the input into three sections: white space (as determined by isspace), initial characters, and unrecognized characters, including a terminating null character. The initial characters may comprise an optional sign character, 0x or 0X, when base is 16, and those letters and digits which represent an integer with a radix of base. The letters (a-z or A-Z) are assigned the values 10 to 35 and are permitted only when those values are less than the value of base.

If base is zero, 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 is 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. The ERANGE value is stored in errno in the case of either overflow or underflow.

Example

```
#include <stdlib.h>
#define base 10

char *rem;
long long int i;

i = strtoll("2345.5", &rem, base);
    /* i=2345, rem=".5" */
```

See Also

atoll, strtofxfx, strtoul

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 `unsignedlongint` the value represented by the string `nptr`. If `endptr` is not a null pointer, `strtoul` stores a pointer to the unconverted remainder in `*endptr`.

The `strtoul` function breaks down the input into three sections:

- Whitespace (as determined by `isspace`)

- Initial characters

- Unrecognized characters including a terminating null character

The initial characters may comprise an optional sign character, `0x` or `0X`, when `base` is 16, and those letters and digits which represent an integer with a radix of `base`. The letters (`a-z` or `A-Z`) are assigned the values 10 to 35 and are permitted only when those values are less than the value of `base`.

If `base` is zero, 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 is 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. The `ERANGE` value 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

strtoull

Convert string to unsigned long long integer

Synopsis

```
#include <stdlib.h>

unsigned long long int strtoull(const char *nptr, char **endptr, int base);
```

Description

The `strtoull` function returns as an `unsigned long long int`, the value represented by the string `nptr`. If `endptr` is not a null pointer, `strtoull` stores a pointer to the unconverted remainder in `*endptr`.

The `strtoull` function breaks down the input into three sections:

- Whitespace (as determined by `isspace`)
- Initial characters
- Unrecognized characters including a terminating null character

The initial characters may comprise an optional sign character, `0x` or `0X`, when `base` is 16, and those letters and digits which represent an integer with a radix of `base`. The letters (`a-z` or `A-Z`) are assigned the values 10 to 35 and are permitted only when those values are less than the value of `base`.

If base is zero, 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 is 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. The ERANGE value is stored in errno in the case of overflow.

Example

```
#include <stdlib.h>
#define base 10

char *rem;
unsigned long long int i;

i = strtoull("2345.5", &rem, base);
    /* i = 2345, rem = ".5" */
```

See Also

atoll, strtofxfx, strtoll

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. The function places the result in the array pointed to by s1.

If s1 and s2 are transformed and used as arguments to strcmp, the result is 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.

The function returns 1. If this value is n or greater, the result stored in the array pointed to by s1 is indeterminate. The s1 can be a null pointer if n is 0.

NOTE: This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

None

Example

```
#include <string.h>

char string1[50];
strxfrm(string1, "SOMEFUN", 49);
        /* SOMEFUN is copied into string1 */
```

See Also

strcmp, strcoll

tan

Tangent

Synopsis

```
#include <math.h>

float tanf (float x);
double tan (double x);
long double tand (long double x);
fract16 tan_fr16 (fract16 x);
fract32 tan_fr32 (fract32 x);
_Fract tan_fx16 (_Fract x);
long _Fract tan_fx32 (long _Fract x);
```

Description

The tangent functions return the tangent of x. Both the argument x and function results are in radians. The defined domain for the `tanf` function is [-9099, 9099], and for the `tand` function the domain is [-51471.0, 51471.0], ...

The `tan_fr16`, `tan_fr32`, `tan_fx16`, and `tan_fx32` functions are defined for fractional input values between [- $\pi/4$, $\pi/4$]. The output from the function is in the range [-1.0, 1.0].

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The tangent functions return a zero if the input argument is not in the defined domain.

Example

```
#include <math.h>

double y;
y = tan (3.14159/4.0)     /* y = 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.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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 returns the value `(time_t) - 1` if the calendar time is not available.

Example

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

if (time(NULL) == (time_t) -1)
    printf("Calendar time is not available\n");
```

See Also

ctime, gmtime, localtime

tmpfile

Create a temporary file

Synopsis

```
#include <stdio.h>

FILE *tmpfile(void);
```

Description

The `tmpfile` is not thread-safe, and is available only if an application is built with the switch `-full-io`.

The `tmpfile` function creates a temporary file and uses `fopen` to open the file in binary read/write mode (mode = `"wb+"`). The `remove` function is used to delete the file when it is closed or when the application terminates.

If successful, the function returns a pointer to the stream; if the function could not open a temporary file, it returns `NULL`.

> **NOTE:** The implementation of the function uses `tmpnam`. Refer to the function's reference page to see how it creates a file name.

Error Conditions

The function returns a null pointer if it could not open a temporary file.

Example

```
#include <stdio.h>
#include <string.h>
#include <stdfix.h>

FILE *tmp1;
FILE *tmp2;

long fract temp_results1[32768];
long fract temp_results2[32768];

tmp1 = tmpfile();
tmp2 = tmpfile();

if ((tmp1) && (tmp2)) {

    /* Save some temporary calculations */
    fwrite (temp_results1,1,sizeof(temp_results1),tmp1);
    fwrite (temp_results2,1,sizeof(temp_results2),tmp2);
    - - - - - - - - - - - - - - -
    /* Restore temporary calculations */
    rewind (tmp1);
    fread (temp_results1,1,sizeof(temp_results1),tmp1);

    rewind (tmp2);
    fread (temp_results2,1,sizeof(temp_results2),tmp2);

    /* Close (and delete) the temporary files */
    fclose (tmp1);
    fclose (tmp2);
}
```

See Also

fopen, tmpnam, remove

tmpnam

Create a name for a temporary file

Synopsis

```
#include <stdio.h>

char *tmpnam(char *tempname);
```

Description

This function is only available if an application is built with the switch `-full-io`.

The `tmpnam` function generates a file name that can be used as the name of a temporary file. If the argument `tempname` is not a `NULL` pointer, the function assumes that the pointer is to an array of at least `L_tmpnam` characters, and it copies the file name into the array.

The function generates a different file name each time that it is called. In this implementation, the file name generated is of the form:

ctm*NNNNN*.tmp

where *NNNNN* represents a five-digit octal number, starting with 00000 and incrementing through to 77777.

> NOTE: The file name generated is a valid file name that is not the same as the name of an existing file. This implementation ensures that it is unique by calling the `remove` function to delete any existing version of the file.

Files whose names are generated by `tmpnam` are only temporary in the sense that their names are unique-unlike files created by `tmpfile`, they are not removed when the application terminates or they are closed; removing the files created by using names generated by `tmpnam` remains the responsibility of the programmer.

The `tmpnam` function is thread-safe and generates a different file name on an application-wide basis-that is, each thread will effectively share a common copy of the function and its data.

The function returns a pointer to the file name. If the argument `tempname` is a `NULL` pointer then the function returns a pointer to internal static memory that contains the file name; this static memory may be overwritten by a subsequent call to `tmpnam`.

Error Conditions

None

Example

```c
#include <stdio.h>

FILE *open_temp_file(char *filename)
{
   return fopen(tmpnam(filename), "w+");
}

void close_temp_file(FILE * workfp, char *filename)
{
    fclose(workfp);
    remove(filename);
}

FILE *workfp;
char workname[L_tmpnam];

workfp = open_temp_file(workname);
close_temp_file(workfp, workname);
```

See Also

tmpfile, fopen, remove

tolower

Convert from uppercase to lowercase

Synopsis

```c
#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.

The function's behavior is only defined if the argument `c` is either `EOF`, or is equivalent to an `unsigned char`.

NOTE: This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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.

The function's behavior is only defined if the argument `c` is either `EOF`, or is equivalent to an `unsigned char`.

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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` is returned by any subsequent read of `stream` in the reverse order of their pushing.

A successful call to the `ungetc` function clears 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 `stdarg.h` header file 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. The function 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, 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_start, va_end

va_end

Finish processing variable-length list of arguments

Synopsis

```
#include <stdarg.h>

void va_end(va_list ap);
```

Description

The `va_end` macro can only be used after the `va_start` macro has been invoked. A call to `va_end` concludes the processing of a variable length list of arguments that was begun by `va_start`.

Error Conditions

None

See Also

va_arg, va_start

va_start

Initialize processing variable-length list of arguments

Synopsis

```
#include <stdarg.h>

void va_start(va_list ap, parmN);
```

Description

The `va_start` macro starts processing of variable arguments in a function declared to take a variable number of arguments. The first argument to `va_start` is 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

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 as `fprintf` with the exception that instead of being a function which takes a variable number or arguments it is called with an argument list `ap` of type `va_list`, as defined in `stdarg.h`.

If the `vfprintf` function is successful it returns 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 returns 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 `NUL` character is written after the end of the last character written to the output array unless `n` is set to zero, in which case nothing will be written to the output array and the output array may be represented by the `NULL` pointer.

The `vsnprintf` function returns the number of characters that would have been written to the output array `str` if `n` was sufficiently large. The return value does not include the terminating `NUL` character written to the array.

Error Conditions

The `vsnprintf` function returns a negative value if unsuccessful.

Example

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

char *message(char *format, ...)
{
   char *message = NULL;
   int len = 0;
   int r;
   va_list p_vargs;         /* return value from vsnprintf  */

   do {
```

```
      va_start (p_vargs,format);
      r = vsnprintf (message,len,format,p_vargs);
      va_end (p_vargs);
      if (r < 0)            /* formatting error?         */
         abort();
      if (r < len)          /* was complete string written? */
         return message;    /* return with success        */
      message = realloc (message,(len=r+1));
   } while (message != NULL);
   abort();
}
```

See Also

fprintf, snprintf

vsprintf

Format argument list into a character array

Synopsis

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

int vsprintf (char *str, const char *format, va_list args);
```

Description

The vsprintf function formats the variable argument list args according to the argument format, and then writes the output to the array str. The argument format contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to fprintf for a description of the valid format specifiers.

The vsprintf function behaves in the same manner as sprintf with the exception that instead of being a function which takes a variable number of arguments, it is called with an argument list args of type va_list, as defined in stdarg.h..

The vsprintf function returns the number of characters that have been written to the output array str. The return value does not include the terminating NUL character written to the array.

Error Conditions

The vsprintf function returns a negative value if unsuccessful.

Example

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

char filename[128];

char *assign_filename(char *filename_template, ...)
{
   char *message = NULL;
   int r;
   va_list p_vargs;            /* return value from vsprintf */
   va_start (p_vargs,filename_template);
   r = vsprintf(&filename[0], filename_template, p_vargs);
   va_end (p_vargs);
   if (r < 0)                  /* formatting error?        */
      abort();
   return &filename[0];     /* return with success      */
}
```

See Also

fprintf, sprintf, snprintf

5 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. These support functions are in addition to the C/C++ run-time library functions described in C and C++ Run-Time Library Guide. (The library also contains functions called implicitly by the compiler, for example `div32`.)

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

This chapter contains:

- DSP Run-Time Library Guide

  Contains information about the library and provides a description of the DSP header files that are included with this release of the `ccblkfn` compiler.

- DSP Run-Time Library Reference

  Contains the complete reference for each DSP run-time library function provided with this release of the `ccblkfn` compiler.

  NOTE: The complete libdsp library is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

DSP Run-Time Library Guide

The DSP run-time library contains functions that can be called from your source program. This section includes:

- Working With Library Source Code

- Library Attributes

- DSP Header Files

- Measuring Cycle Counts

Working With Library Source Code

The source code for the functions in the DSP run-time library is provided with CCES. By default, the libraries are installed in the directory `Blackfin\lib`, and the source files are installed in `Blackfin\lib\src.` Each function is contained in a separate file. The file name is the name of the function with an `.asm` or `.c` extension. If you do not intend to modify any of the run-time library functions, you may delete this directory and its contents to conserve disk space.

Source code is provided so you can customize specific functions. To modify these files, proficiency in Blackfin assembly language and an understanding of the run-time environment is needed.

Refer to C/C++ Run-Time Model and Environment for more information.

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

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

Library Attributes

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

DSP Header Files

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

Table 5-1: DSP Header Files

Header File	Description
complex.h	Basic complex arithmetic functions (complex.h)
cycle_count.h	Basic cycle counting (cycle_count.h)
cycles.h	Cycle counting with statistics (cycles.h)
filter.h	Filters and transformations (filter.h)
math.h	Math functions (math.h)
matrix.h	Matrix functions (matrix.h)
stats.h	Statistical functions (stats.h)
vector.h	Vector functions (vector.h)
window.h	Window generators (window.h)

complex.h

The `complex.h` header file contains type definitions and basic arithmetic operations for variables of type `complex_float`, `complex_double`, `complex_long_double`, `complex_fract16`, and `complex_fract32`. It also contains definitions and prototypes for the C99 native complex support, which uses the `_Complex` keyword.

The complex functions defined in this header file are listed in the *Complex Functions* table. Functions that operate in the `complex_fract16` and `complex_fract32` data types use saturating arithmetic. The `complex_fract16` data type has 32-bit alignment.

The following structures represent complex numbers in rectangular coordinates:

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

typedef struct
{
    double re;
    double im;
} complex_double;

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

typedef struct
{
    #pragma align 4
    fract16 re;
    fract16 im;
} complex_fract16;

typedef struct
{
    fract32 re;
    fract32 im;
} complex_fract32;
```

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. 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 complex functions) which will avoid use of the type-generic macro.

Details about basic complex arithmetic functions are included in DSP Run-Time Library Reference.

Table 5-2: Complex Functions

Description	*Prototype*

Complex Absolute Value	`double cabs (complex_double a)`
	`float cabsf (complex_float a)`
	`long double cabsd (complex_long_double a)`
	`double cabs (complex double a)`
	`float cabsf (complex float a)`
	`long double cabsl (complex long double a)`
	`fract16 cabs_fr16 (complex_fract16 a)`
	`_Fract cabs_fx_fr16 (complex_fract16 a)`
	`fract32 cabs_fr32 (complex_fract32 a)`
	`_Fract cabs_fx_fr16 (complex_fract16 a)`
	`long _Fract cabs_fx_fr32 (complex_fract32 a)`

Complex Addition	`complex_double cadd (complex_double a,`
	`                      complex_double b)`
	`complex_float caddf (complex_float a,`
	`                     complex_float b)`
	`complex_long_double caddd (complex_long_double a,`
	`                           complex_long_double b)`
	`complex_fract16 cadd_fr16 (complex_fract16 a,`
	`                           complex_fract16 b)`
	`complex_fract32 cadd_fr32 (complex_fract32 a,`
	`                           complex_fract32 b)`

Complex Subtraction	`complex_double csub (complex_double a,`
	`                      complex_double b)`
	`complex_float csubf (complex_float a,`
	`                     complex_float b)`
	`complex_long_double csubd (complex_long_double a,`
	`                           complex_long_double b)`
	`complex_fract16 csub_fr16 (complex_fract16 a,`
	`                           complex_fract16 b)`
	`complex_fract32 csub_fr32 (complex_fract32 a,`
	`                           complex_fract32 b)`

Complex Multiply	`complex_double cmlt (complex_double a,`
	`                      complex_double b)`
	`complex_float cmltf (complex_float a,`
	`                     complex_float b)`
	`complex_long_double cmltd (complex_long_double a,`
	`                           complex_long_double b)`
	`complex_fract16 cmlt_fr16 (complex_fract16 a,`
	`                           complex_fract16 b)`
	`complex_fract32 cmlt_fr32 (complex_fract32 a,`
	`                           complex_fract32 b)`

Table 5-2: Complex Functions (Continued)

Description	Prototype
| Complex Division | ```
complex_double cdiv (complex_double a, complex_double b)
complex_float cdivf (complex_float a, complex_float b)
complex_long_double cdivd (complex_long_double a,
                           complex_long_double b)
complex_fract16 cdiv_fr16 (complex_fract16 a,
                           complex_fract16 b)
complex_fract32 cdiv_fr32 (complex_fract32 a,
                           complex_fract32 b)
``` |
| Get Phase of a Complex Number | ```
double arg (complex_double a)
float argf (complex_float a)
long double argd (complex_long_double a)
double carg (complex double a)
float cargf (complex float a)
long double cargl (complex long double a)
fract16 arg_fr16 (complex_fract16 a)
fract32 arg_fr32 (complex_fract32 a)
_Fract arg_fx_fr16 (complex_fract16 a)
long _Fract arg_fx_fr32 (complex_fract32
a)
``` |
| Complex Conjugate | ```
complex_double conj (complex_double a)
complex_float conjf (complex_float a)
complex_long_double conjd (complex_long_double a)
complex_double conj (complex double a)
complex_float conjf (complex float a)
complex_long_double conjl (complex long double a)
complex_fract16 conj_fr16 (complex_fract16 a)
complex_fract32 conj_fr32 (complex_fract32 a)
``` |
| Convert Cartesian to Polar Coordinates | ```
double cartesian (complex_double a, double* phase)
float cartesianf (complex_float a, float* phase)
long double cartesiand (complex_long_double a,
                        long double* phase)
fract16 cartesian_fr16 (complex_fract16 a,
                        fract16* phase)
fract32 cartesian_fr32 (complex_fract32 a,
                        fract32* phase)
_Fract cartesian_fx_fr16 (complex_fract16 a,
                          _Fract* phase)
long _Fract cartesian_fx_fr32 (complex_fract32 a,
                               long _Fract* phase)
``` |
| Convert Polar to Cartesian Coordinates | ```
complex_double polar (double mag, double phase)
complex_float polarf (float mag, float phase)
complex_long_double polard (long double mag,
                            long double phase)
complex_fract16 polar_fr16 (fract16 mag,
                            fract16 phase)
complex_fract32 polar_fr32 (fract32 mag,
                            fract32 phase)
complex_fract16 polar_fx_fr16(_Fract mag,
                              _Fract phase)
``` |

Table 5-2: Complex Functions (Continued)

Description	*Prototype*
| | ```
complex_fract32 polar_fx_fr32 (long _Fract mag,
                               long _Fract
phase)
``` |
| Complex Exponential | ```
complex_double cexp (double a)
complex_long_double cexpd (long double a)
complex_float cexpf (float a)
complex double cexp (double a)
complex long double cexpl (long double a)
complex float cexpf (float a)
``` |
| Imaginary Part | ```
double cimag (complex double a)
long double cimagl (complex long double a)
float cimagf (complex float a)
``` |
| Real Part | ```
double creal (complex double a)
long double creall (complex long double a)
float crealf (complex float a)
``` |
| Normalization | ```
complex_double norm (complex_double a)
complex_long_double normd (complex_long_double a)
complex_float normf (complex_float a)
``` |

cycle_count.h

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 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 the minimum, average, and maximum number of cycles used. The facilities available via this header file are described in Measuring Cycle Counts.

filter.h

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

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

The library supports three different sets of FFT function. Each set consists of an FFT function for a complex input signal, a function for a real input signal, and a function that computes the inverse of an FFT. The FFT functions are available for both the `fract16` and `fract32` data types. The first set of functions are radix-2 FFT functions that support three different forms of scaling, The second set are optimized mixed-radix functions that only support static scaling and the third set of functions compute a 2-dimensional FFT. The number of points in an FFT is specified as

a function parameter and must be a power of 2. The twiddle table for the FFT functions is supplied as a separate argument and is normally calculated once during program initialization. All FFT functions have also a stride argument as function parameter to facilitate sharing of twiddle tables between different sized FFTs.

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

Various pre-computed twiddle tables are declared and may be defined in ROM on parts that have utility ROM support, or in RAM.

An FFT magnitude function is also provided that computes the normalized power spectrum of an FFT.

The functions defined in this header file are listed in the *Filter Library* and *Transformational Functions* tables and described in DSP Run-Time Library Reference.

Table 5-3: Filter Library

Description	*Prototype*

Finite Impulse Response Filter	```void fir_fr16 (const fract16 input[], fract16 output[],```
	 ``` int length, fir_state_fr16 *filter_state)```
	 ```void fir_fx16 (const _Fract input[], _Fract output[],```
	 ``` int length, fir_state_fx16 *filter_state)```
	 ```void fir_fr32 (const fract32 input[], fract32 output[],```
	 ``` int length, fir_state_fr32 *filter_state)```
	 ```void fir_fx32 (const long _Fract input[], long _Fract output[],```
	 ``` int length, fir_state_fx32 *filter_state)```

Infinite Impulse Response Filter	```void iir_fr16 (const fract16 input[], fract16 output[],```
	 ``` int length, iir_state_fr16 *filter_state)```
	 ```void iir_fx16 (const _Fract input[], _Fract output[],```
	 ``` int length, iir_state_fx16 *filter_state)```
	 ```void iir_fr32 (const fract32 input[], fract32 output[],```
	 ``` int length, iir_state_fr32 *filter_state)```
	 ```void iir_fx32 (const long _Fract input[], long _Fract output[],```
	 ``` int length, iir_state_fx32 *filter_state)```

Direct Form I Infinite Response Filter	```void iirdf1_fr16 (const fract16 input[], fract16 output[],```
	 ``` int length, iirdf1_state_fr16 *filter_state)```
	 ```void iirdf1_fx16 (const _Fract input[], _Fract output[],```
	 ``` int length, iirdf1_state_fx16 *filter_state)```
	 ```void iirdf1_fr32 (const fract32 input[], fract32 output[],```
	 ``` int length, iirdf1_state_fr32 *filter_state)```
	 ```void iirdf1_fx32 (const long _Fract input[], long _Fract output[],```
	 ``` int length, iirdf1_state_fx32 *filter_state)```

FIR Decimation Filter	```void fir_decima_fr16 (const fract16 input[], fract16 output[],```
	 ``` int length, fir_state_fr16 *filter_state)```
	 ```void fir_decima_fx16 (const _Fract input[], _Fract output[],```
	 ``` int length, fir_state_fx16 *filter_state)```
	 ```void fir_decima_fr32 (const fract32 input[], fract32 output[],```

Table 5-3: Filter Library (Continued)

Description	Prototype
| | ```
                         int length, fir_state_fr32 *filter_state)
void fir_decima_fx32 (const long _Fract input[],
                      long _Fract output[],
                      int length, fir_state_fx32 *filter_state)
``` |
| FIR Interpolation Filter | ```
void fir_interp_fr16 (const fract16 input[], fract16 output[],
                      int length, fir_state_fr16 *filter_state)
void fir_interp_fx16 (const _Fract input[], _Fract output[],
                      int length, fir_state_fx16 *filter_state)
void fir_interp_fr32 (const fract32 input[], fract32 output[],
                      int length, fir_state_fr32 *filter_state)
void fir_interp_fx32 (const long _Fract input[],
                      long _Fract output[],
                      int length, fir_state_fx32 *filter_state)
``` |
| Complex Finite Impulse Response Filter | ```
void cfir_fr16 (const complex_fract16 input[],
                complex_fract16 output[],
                int length, cfir_state_fr16 *filter_state)
void cfir_fr32 (const complex_fract32 input[],
                complex_fract32 output[],
                int length, cfir_state_fr32 *filter_state)
``` |
| Convert Coefficients for DF1 IIR | ```
void coeff_iirdf1_fr16 (const float acoeff[],
                        const float bcoeff[ ],
                        fract16 coeff[], int nstages)
void coeff_iirdf1_fx16 (const float acoeff[],
                        const float bcoeff[ ],
                        _Fract coeff[], int nstages)
void coeff_iirdf1_fr32 (const long double acoeff[],
                        const long double bcoeff[ ],
                        fract32 coeff[], int nstages)
void coeff_iirdf1_fx32 (const long double acoeff[],
                        const long double bcoeff[ ],
                        long _Fract coeff[], int nstages)
``` |

Table 5-4: Transformational Functions

Description	Prototype
Fast Fourier Transforms	
| Generate FFT Twiddle Factors for Radix-2 FFT | ```
void twidfftrad2_fr16 (complex_fract16 twiddle_table[],
                       int fft_size)
void twidfftrad2_fr32 (complex_fract32 twiddle_table[],
                       int fft_size)
``` |
| Generate FFT Twiddle Factors for 2-D FFT | ```
void twidfft2d_fr16 (complex_fract16 twiddle_table[], int fft_size)
void twidfft2d_fr32 (complex_fract32 twiddle_table[], int fft_size)
``` |
| Generate FFT Twiddle Factors for Optimized FFT | ```
void twidfftf_fr16 (complex_fract16 twiddle_table[], int fft_size)
void twidfftf_fr32 (complex_fract32 twiddle_table[], int fft_size)
``` |

Table 5-4: Transformational Functions (Continued)

Description	Prototype

Pre-computed FFT Twiddle Factors	```const complex_fract16 twidfftf_fr16_8k_table[]```
	```const complex_fract32 twidfftf_fr32_4k_table[]```
	```const complex_fract16 twidfftrad_fr16_8k_table[]```
	```const complex_fract32 twidfftrad_fr32_4k_table[]```

FFT Magnitude	```void fft_magnitude_fr16(const complex_fract16 input[],```
	``` fract16 output[],```
	``` int fft_size, int block_exponent,```
	``` int mode)```
	```void fft_magnitude_fr32 (const complex_fract32 input[],```
	``` fract32 output[],```
	``` int fft_size, int block_exponent,```
	``` int mode)```

N Point Radix-2 Complex Input FFT	```void cfft_fr16 (const complex_fract16 *input,```
	``` complex_fract16 *output,```
	``` const complex_fract16 *twiddle_table,```
	``` int twiddle_stride, int fft_size,```
	``` int *block_exponent, int scale_method)```
	```void cfft_fr32 (const complex_fract32 *input,```
	``` complex_fract32 *output,```
	``` const complex_fract32 *twiddle_table,```
	``` int twiddle_stride,```
	``` int fft_size, int *block_exponent,```
	``` int scale_method)```

N Point Radix-2 Real Input FFT	```void rfft_fr16 (const fract16 *input,```
	``` complex_fract16 *output,```
	``` const complex_fract16 *twiddle_table,```
	``` int twiddle_stride,```
	``` int fft_size, int *block_exponent,```
	``` int scale_method)```
	```void rfft_fx_fr16 (const _Fract *input, complex_fract16 *output,```
	``` const complex_fract16 *twiddle_table,```
	``` int twiddle_stride,```
	``` int fft_size, int *block_exponent,```
	``` int scale_method)```
	```void rfft_fr32 (const fract32 *input,```
	``` complex_fract32 *output,```
	``` const complex_fract32 *twiddle_table,```
	``` int twiddle_stride,```
	``` int fft_size, int *block_exponent,```
	``` int scale_method)```
	```void rfft_fx_fr32 (const long _Fract *input,```
	``` complex_fract32 *output,```
	``` const complex_fract32 *twiddle_table,```
	``` int twiddle_stride,```
	``` int fft_size, int *block_exponent,```
	``` int scale_method)```

N Point Radix-2 Inverse FFT	```void ifft_fr16 (const complex_fract16 *input,```
	``` complex_fract16 *output,```
	``` const complex_fract16 *twiddle_table,```

Table 5-4: Transformational Functions (Continued)

Description	Prototype

	```int twiddle_stride,```
	 ```int fft_size, int *block_exponent,```
	 ```int scale_method)```
	 ```void ifft_fr32 (const complex_fract32 *input,```
	 ```complex_fract32 *output,```
	 ```const complex_fract32 *twiddle_table,```
	 ```int twiddle_stride,```
	 ```int fft_size, int *block_exponent,```
	 ```int scale_method)```

Fast N point Radix-4 Complex Input FFT	```void cfftf_fr16 (const complex_fract16 *input,```
	 ```complex_fract16 *output,```
	 ```const complex_fract16 *twiddle_table,```
	 ```int twiddle_stride, int fft_size)```
	 ```void cfftf_fr32 (const complex_fract32 *input,```
	 ```complex_fract32 *output,```
	 ```const complex_fract32 *twiddle_table,```
	 ```int twiddle_stride, int fft_size)```

Fast N point Mixed-Radix Inverse Input FFT	```void ifftf_fr16 (const complex_fract16 *input,```
	 ```complex_fract16 *output,```
	 ```const complex_fract16 *twiddle_table,```
	 ```int twiddle_stride, int fft_size)```
	 ```void ifftf_fr32 (const complex_fract32 *input,```
	 ```complex_fract32 *output,```
	 ```const complex_fract32 *twiddle_table,```
	 ```int twiddle_stride,```
	 ```int fft_size)```

Fast N point Mixed-Radix Real Input FFT	```void rfftf_fr16 (const complex_fract16 *input,```
	 ```complex_fract16 *output,```
	 ```const complex_fract16 *twiddle_table,```
	 ```int twiddle_stride, int fft_size)```
	 ```void rfftf_fx_fr16 (const _Fract *input,```
	 ```complex_fract16 *output,```
	 ```const complex_fract16 *twiddle_table,```
	 ```int twiddle_stride, int fft_size)```
	 ```void rfftf_fr32 (const complex_fract32 *input,```
	 ```complex_fract32 *output,```
	 ```const complex_fract32 *twiddle_table,```
	 ```int twiddle_stride, int fft_size)```
	 ```void rfftf_fx_fr32 (const long _Fract *input,```
	 ```complex_fract32 *output,```
	 ```const complex_fract32 *twiddle_table,```
	 ```int twiddle_stride, int fft_size)```

NxN Point 2-D Complex Input FFT	```void cfft2d_fr16 (const complex_fract16 *input,```
	 ```complex_fract16 *temp,```
	 ```complex_fract16 *output,```
	 ```const complex_fract16 *twiddle_table,```
	 ```int twiddle_stride, int fft_size,```
	 ```int block_exponent, int scale_method)```
	 ```void cfft2d_fr32 (const complex_fract32 *input,```

Table 5-4: Transformational Functions (Continued)

Description	Prototype
| | ```
          complex_fract32 *temp,
          complex_fract32 *output,
          const complex_fract32 *twiddle_table,
          int twiddle_stride, int fft_size)
``` |
| NxN Point 2-D Real Input FFT | ```
void rfft2d_fr16 (const fract16 *input,
          complex_fract16 *temp,
          complex_fract16 *output,
          const complex_fract16 *twiddle_table,
          int twiddle_stride, int fft_size,
          int block_exponent,
          int scale_method)
void rfft2d_fx_fr16 (const _Fract *input,
          complex_fract16 *temp,
          complex_fract16 *output,
          const complex_fract16 *twiddle_table,
          int twiddle_stride, int fft_size,
          int block_exponent, int scale_method)
void rfft2d_fr32 (const fract32 *input,
          complex_fract32 *temp,
          complex_fract32 *output,
          const complex_fract32 *twiddle_table,
          int twiddle_stride, int fft_size)
void rfft2d_fx_fr32 (const long _Fract *input,
          complex_fract32 *temp,
          complex_fract32 *output,
          const complex_fract32 *twiddle_table,
          int twiddle_stride, int fft_size)
``` |
| NxN Point 2-D Inverse FFT | ```
void ifft2d_fr16 (const complex_fract16 *input,
          complex_fract16 *temp,
          complex_fract16 *output,
          const complex_fract16 *twiddle_table,
          int twiddle_stride, int fft_size,
          int block_exponent, int scale_method)
void ifft2d_fr32 (const complex_fract32 *input,
          complex_fract32 *temp,
          complex_fract32 *output,
          const complex_fract32 *twiddle_table,
          int twiddle_stride, int fft_size)
``` |
| *Convolutions* | |
| Convolution | ```
void convolve_fr16 (const fract16 input_x[], int length_x,
          const fract16 input_y[], int length_y,
          fract16 output[])
void convolve_fr32 (const fract32 input_x[], int length_x,
          const fract32 input_y[], int length_y,
          fract32 output[])
void convolve_fx16 (const _Fract input_x[], int length_x,
          const _Fract input_y[], int length_y,
          _Fract output[])
void convolve_fx32 (const long _Fract input_x[], int length_x,
``` |

Table 5-4: Transformational Functions (Continued)

Description	Prototype
	`const long _Fract input_y[], int length_y,` `long _Fract output[])`

2-D Convolution	`void conv2d_fr16 (const fract16 *input_x, int rows_x,`
	`                int columns_x,`
	`                const fract16 *input_y, int rows_y,`
	`                int columns_y, fract16 *output)`
	`void conv2d_fx16 (const _Fract *input_x, int rows_x,`
	`                int columns_x,`
	`                const _Fract *input_y, int rows_y,`
	`                int columns_y, _Fract *output)`
	`void conv2d_fr32 (const fract32 *input_x, int rows_x,`
	`                int columns_x,`
	`                const fract32 *input_y, int rows_y,`
	`                int columns_y, fract32 *output)`
	`void conv2d_fx32 (const long _Fract *input_x, int rows_x,`
	`                int columns_x,`
	`                const long _Fract *input_y, int rows_y,`
	`                int columns_y, long _Fract *output)`

2-D Convolution 3x3 Matrix	`void conv2d3x3_fr16 (const fract16 *input_x, int rows_x,`
	`                int columns_x,`
	`                const fract16 input_y [],`
	`                fract16 *output)`
	`void conv2d3x3_fx16 (const _Fract *input_x, int rows_x,`
	`                int columns_x,`
	`                const _Fract input_y [],`
	`                _Fract *output)`
	`void conv2d3x3_fr32 (const fract32 *input_x, int rows_x,`
	`                int columns_x,`
	`                const fract32 input_y [],`
	`                fract32 *output)`
	`void conv2d3x3_fx32 (const long _Fract *input_x, int rows_x,`
	`                int columns_x,`
	`                const long _Fract input_y [],`
	`                long _Fract *output)`

Compression/Expansion	
A-law compression	`void a_compress (const short input[], short output[], int length)`
A-law expansion	`void a_expand (const short input[], short output[], int length)`
μ-law compression	`void mu_compress (const short input[], short output[], int length)`
μ-law expansion	`void mu_expand (const short input[], short output[], int length)`

math.h

The standard math functions have been augmented by implementations for the `float` and `long double` data types, and in some cases, for the `fract16` and `fract32` data types and the Embedded C data types, `_Fract` and `long _Fract`.

The *Math Library* table summarizes the functions defined by the `math.h` header file. Descriptions of these functions are given under the name of the `double` version in the C Run-Time Library Reference chapter.

The `math.h` header file also provides prototypes for additional math functions (`clip`, `copysign`, `max`, and `min`), and an integer function (`countones`). These functions are described in DSP Run-Time Library Reference.

Table 5-5: Math Library

Description	*Prototype*

Absolute Value	`double fabs (double x)`
	`float fabsf (float x)`
	`long double fabsd (long double x)`

Anti-log	`double alog (double x)`
	`float alogf (float x)`
	`long double alogd (long double x)`

Base 10 Anti-log	`double alog10 (double x)`
	`float alog10f (float x)`
	`long double alog10d (long double x)`

Arc Cosine	`double acos (double x)`
	`float acosf (float x)`
	`long double acosd (long double x)`
	`fract16 acos_fr16 (fract16 x)`
	`_Fract acos_fx16 (_Fract x)`
	`fract32 acos_fr32 (fract32 x)`
	`long _Fract acos_fx32 (long _Fract x)`

Arc Sine	`double asin (double x)`
	`float asinf (float x)`
	`long double asind (long double x)`
	`fract16 asin_fr16 (fract16 x)`
	`_Fract asin_fx16 (_Fract x)`
	`fract32 asin_fr32 (fract32 x)`
	`long _Fract asin_fx32 (long _Fract x)`

Arc Tangent	`double atan (double x)`
	`float atanf (float x)`
	`long double atand (long double x)`
	`fract16 atan_fr16 (fract16 x)`
	`_Fract atan_fx16 (_Fract x)`
	`fract32 atan_fr32 (fract32 x)`
	`long _Fract atan_fx32 (long _Fract x)`

Arc Tangent of Quotient	`double atan2 (double y, double x)`
	`float atan2f (float y, float x)`
	`long double atan2d (long double y, long double x)`
	`fract16 atan2_fr16 (fract16 y, fract16 x)`
	`_Fract atan2_fx16 (_Fract y, _Fract x)`
	`fract32 atan2_fr32 (fract32 y, fract32 x)`
	`long _Fract atan2_fx32 (long _Fract y, long _Fract x)`

Table 5-5: Math Library (Continued)

Description	Prototype

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

Cosine	```double cos (double x)```
	```float cosf (float x)```
	```long double cosd (long double x)```
	```fract16 cos_fr16 (fract16 x)```
	```_Fract cos_fx16 (_Fract x)```
	```fract32 cos_fr32 (fract32 x)```
	```long _Fract cos_fx32 (long _Fract x)```

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

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

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

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

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

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

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

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

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

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

Table 5-5: Math Library (Continued)

Description	Prototype
| Logarithm Base 10 | ```
double log10 (double x)
float log10f (float x)
long double log10d (long double x)
``` |
| Get Fraction and Integer | ```
double modf (double x, double *i)
float modff (float x, float *i)
long double modfd (long double x, long double *i)
``` |
| Power | ```
double pow (double x, double y)
float powf (float x, float y)
long double powd (long double x, long double y)
``` |
| Reciprocal Square Root | ```
double rsqrt (double x)
float rsqrtf (float x)
long double rsqrtd (long double x)
``` |
| Sine | ```
double sin (double x)
float sinf (float x)
long double sind (long double x)
fract16 sin_fr16 (fract16 x)
_Fract sin_fx16 (_Fract x)
fract32 sin_fr32 (fract32 x)
long _Fract sin_fx32 (long _Fract x)
``` |
| Hyperbolic Sine | ```
double sinh (double x)
float sinhf (float x)
long double sinhd (long double x)
``` |
| Square Root | ```
double sqrt (double x)
float sqrtf (float x)
long double sqrtd (long double x)
fract16 sqrt_fr16 (fract16 x)
fract32 sqrt_fr32 (fract32 x)
_Fract sqrt_fx16 (_Fract x)
long _Fract sqrt_fx32 (long _Fract x)
``` |
| Tangent | ```
double tan (double x)
float tanf (float x)
long double tand (long double x)
fract16 tan_fr16 (fract16 x)
fract32 tan_fr32 (fract32 x)
_Fract tan_fx16 (_Fract x)
long _Fract tan_fx32 (long _Fract x)
``` |
| Hyperbolic Tangent | ```
double tanh (double x)
float tanhf (float x)
long double tanhd (long double x)
``` |

matrix.h

The matrix.h header file contains matrix functions for operating on real and complex matrices, both matrix-scalar and matrix-matrix operations. See complex.h for definitions of the complex types.

CCES 2.9.0 C/C++ Compiler and Library Manual for Blackfin Processors

The matrix functions defined in the `matrix.h` header file are listed in the *Matrix Functions* table. Matrix functions that operate on the `fract16`, `fract32`, `complex_fract16` and `complex_fract32` data types, and on the Embedded C data types `_Fract` and `long _Fract`, use saturating arithmetic.

Table 5-6: Matrix Functions

Description	Prototype

Real Matrix + Scalar Addition	`void matsadd (const double *matrix, double scalar,`
	`              int rows, int columns, double *out)`
	`void matsaddf (const float *matrix, float scalar,`
	`               int rows, int columns, float *out)`
	`void matsaddd (const long double *matrix, long double scalar,`
	`               int rows, int columns, long double *out)`
	`void matsadd_fr16 (const fract16 *matrix, fract16 scalar,`
	`                   int rows, int columns, fract16 *out)`
	`void matsadd_fr32 (const fract32 *matrix, fract32 scalar,`
	`                   int rows, int columns, fract32 *out)`
	`void matsadd_fx16 (const _Fract *matrix, _Fract scalar,`
	`                   int rows, int columns, _Fract *out)`
	`void matsadd_fx32 (const long _Fract *matrix, long _Fract scalar,`
	`                   int rows, int columns, long _Fract *out)`

Real Matrix - Scalar Subtraction	`void matssub (const double *matrix, double scalar,`
	`              int rows, int columns, double *out)`
	`void matssubf (const float *matrix, float scalar,`
	`               int rows, int columns, float *out)`
	`void matssubd (const long double *matrix, long double scalar,`
	`               int rows, int columns, long double *out)`
	`void matssub_fr16 (const fract16 *matrix, fract16 scalar,`
	`                   int rows, int columns, fract16 *out)`
	`void matssub_fr32 (const fract32 *matrix, fract32 scalar,`
	`                   int rows, int columns, fract32 *out)`
	`void matssub_fx16 (const _Fract *matrix, _Fract scalar,`
	`                   int rows, int columns, _Fract *out)`
	`void matssub_fx32 (const long _Fract *matrix,`
	`                   long _Fract scalar, int rows,`
	`                   int columns, long _Fract *out)`

Real Matrix * Scalar Multiplication	`void matsmlt (const double *matrix, double scalar,`
	`              int rows, int columns, double *out)`
	`void matsmltf (const float *matrix, float scalar,`
	`               int rows, int columns, float *out)`
	`void matsmltd (const long double *matrix,`
	`               long double scalar, int rows,`
	`               int columns, long double *out)`
	`void matsmlt_fr16 (const fract16 *matrix, fract16 scalar,`
	`                   int rows, int columns, fract16 *out)`
	`void matsmlt_fr32 (const fract32 *matrix, fract32 scalar,`
	`                   int rows, int columns, fract32 *out)`
	`void matsmlt_fx16 (const _Fract *matrix, _Fract scalar,`
	`                   int rows, int columns, _Fract *out)`
	`void matsmlt_fx32 (const long _Fract *matrix,`
	`                   long _Fract scalar, int rows,`
	`                   int columns, long _Fract *out)`

Table 5-6: Matrix Functions (Continued)

Description	Prototype
Real Matrix + Matrix Addition	```void matmadd (const double *matrix_a, const double *matrix_b,
int rows, int columns, double *out)
void matmaddf (const float *matrix_a, const float *matrix_b,
int rows, int columns, float *out)
void matmaddd (const long double *matrix_a,
const long double *matrix_b, int rows,
int columns, long double *out)
void matmadd_fr16 (const fract16 *matrix_a,
const fract16 *matrix_b, int rows,
int columns, fract16 *out)
void matmadd_fr32 (const fract32 *matrix_a,
const fract32 *matrix_b, int rows,
int columns, fract32 *out)
void matmadd_fx16 (const _Fract *matrix_a,
const _Fract *matrix_b, int rows,
int columns, _Fract *out)
void matmadd_fx32 (const long _Fract *matrix_a,
const long _Fract *matrix_b,
int rows, int columns, long _Fract *out)```
Real Matrix - Matrix Subtraction	```void matmsub (const double *matrix_a, const double *matrix_b,
int rows, int columns, double *out)
void matmsubf (const float *matrix_a, const float *matrix_b,
int rows, int columns, float *out)
void matmsubd (const long double *matrix_a, const long double *matrix_b,
int rows, int columns, long double *out)
void matmsub_fr16 (const fract16 *matrix_a, const fract16 *matrix_b,
int rows, int columns, fract16 *out)
void matmsub_fr32 (const fract32 *matrix_a, const fract32 *matrix_b,
int rows, int columns, fract32 *out)
void matmsub_fx16 (const _Fract *matrix_a, const _Fract *matrix_b,
int rows, int columns, _Fract *out)
void matmsub_fx32 (const long _Fract *matrix_a,
const long _Fract *matrix_b,
int rows, int columns, long _Fract *out)```
Real Matrix * Matrix Multiplication	```void matmmlt (const double *matrix_a, int rows_a, int columns_a,
const double *matrix_b, int columns_b, double *out)
void matmmltf (const float *matrix_a, int rows_a, int columns_a,
const float *matrix_b, int columns_b, float *out)
void matmmltd (const long double *matrix_a, int rows_a, int columns_a,
const long double *matrix_b, int columns_b,
long double *out)
void matmmlt_fr16 (const fract16 *matrix_a, int rows_a, int columns_a,
const fract16 *matrix_b, int columns_b, fract16 *out)
void matmmlt_fr32 (const fract32 *matrix_a, int rows_a, int columns_a,
const fract32 *matrix_b, int columns_b, fract32 *out)
void matmmlt_fx16 (const _Fract *matrix_a, int rows_a, int columns_a,
const _Fract *matrix_b, int columns_b, _Fract *out)
void matmmlt_fx32 (const long _Fract *matrix_a, int rows_a,
int columns_a, const long _Fract *matrix_b,
int columns_b, long _Fract *out)```

Table 5-6: Matrix Functions (Continued)

Description	*Prototype*
| Complex Matrix + Scalar Addition | ```
void cmatsadd (const complex_double *matrix, complex_double scalar,
               int rows, int columns, complex_double *out)
void cmatsaddf (const complex_float *matrix, complex_float scalar,
                int rows, int columns, complex_float *out)
void cmatsaddd (const complex_long_double *matrix,
                complex_long_double scalar, int rows,
                int columns, complex_long_double *out)
void cmatsadd_fr16 (const complex_fract16 *matrix,
                    complex_fract16 scalar, int rows,
                    int columns, complex_fract16 *out)
void cmatsadd_fr32 (const complex_fract32 *matrix,
                    complex_fract32 scalar, int rows,
                    int columns, complex_fract32 *out)
``` |
| Complex Matrix - Scalar Subtraction | ```
void cmatssub (const complex_double *matrix, complex_double scalar,
               int rows, int columns, complex_double *out)
void cmatssubf (const complex_float *matrix, complex_float scalar,
                int rows, int columns, complex_float *out)
void cmatssubd (const complex_long_double *matrix,
                complex_long_double scalar, int rows,
                int columns, complex_long_double *out)
void cmatssub_fr16 (const complex_fract16 *matrix,
                    complex_fract16 scalar, int rows,
                    int columns, complex_fract16 *out)
void cmatssub_fr32 (const complex_fract32 *matrix,
                    complex_fract32 scalar, int rows,
                    int columns, complex_fract32 *out)
``` |
| Complex Matrix * Scalar Multiplication | ```
void cmatsmlt (const complex_double *matrix, complex_double scalar,
               int rows, int columns, complex_double *out)
void cmatsmltf (const complex_float *matrix, complex_float scalar,
                int rows, int columns, complex_float *out)
void cmatsmltd (const complex_long_double *matrix,
                complex_long_double scalar, int rows,
                int columns, complex_long_double *out)
void cmatsmlt_fr16 (const complex_fract16 *matrix,
                    complex_fract16 scalar, int rows,
                    int columns, complex_fract16 *out)
void cmatsmlt_fr32 (const complex_fract32 *matrix,
                    complex_fract32 scalar, int rows,
                    int columns, complex_fract32 *out)
``` |
| Complex Matrix + Matrix Addition | ```
void cmatmadd (const complex_double *matrix_a,
               const complex_double *matrix_b, int rows,
               int columns, complex_double *out)
void cmatmaddf (const complex_float *matrix_a,
                const complex_float *matrix_b, int rows,
                int columns, complex_float *out)
void cmatmaddd (const complex_long_double *matrix_a,
                const complex_long_double *matrix_b, int rows,
                int columns, complex_long_double *out)
void cmatmadd_fr16 (const complex_fract16 *matrix_a,
                    const complex_fract16 *matrix_b,
``` |

Table 5-6: Matrix Functions (Continued)

Description	Prototype
| | ```
                            int rows, int columns, complex_fract16 *out)
void cmatmadd_fr32 (const complex_fract32 *matrix_a,
                    const complex_fract32 *matrix_b, int rows,
                    int columns, complex_fract32 *out)
``` |
| Complex Matrix -Matrix Subtraction | ```
void cmatmsub (const complex_double *matrix_a,
               const complex_double *matrix_b, int rows,
               int columns, complex_double *out)
void cmatmsubf (const complex_float *matrix_a,
                const complex_float *matrix_b, int rows,
                int columns, complex_float *out)
void cmatmsubd (const complex_long_double *matrix_a,
                const complex_long_double *matrix_b, int rows,
                int columns, complex_long_double *out)
void cmatmsub_fr16 (const complex_fract16 *matrix_a,
                    const complex_fract16 *matrix_b, int rows,
                    int columns, complex_fract16 *out)
void cmatmsub_fr32 (const complex_fract32 *matrix_a,
                    const complex_fract32 *matrix_b,
                    int rows, int columns, complex_fract32 *out)
``` |
| Complex Matrix * Matrix Multiplication | ```
void cmatmmlt (const complex_double *matrix_a, int rows_a,
               int columns_a, const complex_double *matrix_b,
               int columns_b, complex_double *out)
void cmatmmltf (const complex_float *matrix_a, int rows_a,
                int columns_a, const complex_float *matrix_b,
                int columns_b, complex_float *out)
void cmatmmltd (const complex_long_double *matrix_a, int rows_a,
                int columns_a, const complex_long_double *matrix_b,
                int columns_b, complex_long_double *out)
void cmatmmlt_fr16 (const complex_fract16 *matrix_a, int rows_a,
                    int columns_a, const complex_fract16 *matrix_b,
                    int columns_b, complex_fract16 *out)
void cmatmmlt_fr32 (const complex_fract32 *matrix_a, int rows_a,
                    int columns_a, const complex_fract32 *matrix_b,
                    int columns_b, complex_fract32 *out)
``` |
| Transpose | ```
void transpm (const double *matrix, int rows,
              int columns, double *out)
void transpmf (const float *matrix, int rows,
               int columns, float *out)
void transpmd (const long double *matrix, int rows,
               int columns, long double *out)
void transpm_fr16 (const fract16 *matrix, int rows,
                   int columns, fract16 *out)
void transpm_fr32 (const fract32 *matrix, int rows,
                   int columns, fract32 *out)
void transpm_fx16 (const _Fract *matrix, int rows,
                   int columns, _Fract *out)
void transpm_fx32 (const long _Fract *matrix, int rows,
                   int columns, long _Fract *out)
``` |

Table 5-6: Matrix Functions (Continued)

Description	Prototype
| Complex Transpose | ```
void ctranspm (const complex_double *matrix, int rows,
               int columns, complex_double *out)
void ctranspmf (const complex_float *matrix, int rows,
                int columns, complex_float *out)
void ctranspmd (const complex_long_double *matrix, int rows,
                int columns, complex_long_double *out)
void ctranspm_fr16 (const complex_fract16 *matrix, int rows,
                    int columns, complex_fract16 *out)
void ctranspm_fr32 (const complex_fract32 *matrix,
                    int rows int columns, complex_fract32 *out)
``` |

In most of the function prototypes:

`*matrix_a`	Is a pointer to input matrix `matrix_a[][]`
`*matrix_b`	Is a pointer to input matrix `matrix_b[][]`
`scalar`	Is an input scalar
`rows`	Is the number of rows
`columns`	Is the number of columns
`*out`	Is a pointer to output matrix `out[][]`

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

The functions described by this header assume that input array arguments are constant; that is, their contents do not change during the course of the routine. In particular, this means the input arguments do not overlap with any output argument.

stats.h

The statistical functions defined in the `stats.h` header file are listed in the *Statistical Functions* table and are described in DSP Run-Time Library Reference.

Table 5-7: Statistical Functions

Description	Prototype
| Autocoherence | ```
void autocohf (const float samples[], int sample_length,
               int lags, float out[])
void autocoh (const double samples[], int sample_length,
              int lags, double out[])
void autocohd (const long double samples[], int sample_length,
               int lags, long double out[])
void autocoh_fr16 (const fract16 samples[], int sample_length,
                   int lags, fract16 out[])
``` |

Table 5-7: Statistical Functions (Continued)

Description	Prototype

	```void autocoh_fr32 (const fract32 samples[], int sample_length,```
	 ``` int lags, fract32 out[])```
	 ```void autocoh_fx16 (const _Fract samples[], int sample_length,```
	 ``` int lags, _Fract out[])```
	 ```void autocoh_fx32 (const long _Fract samples[], int sample_length,```
	 ``` int lags, long _Fract out[])```

Autocorrelation	```void autocorrf (const float samples[], int sample_length,```
	 ``` int lags, float out[])```
	 ```void autocorr (const double samples[], int sample_length,```
	 ``` int lags, double out[])```
	 ```void autocorrd (const long double samples[], int sample_length,```
	 ``` int lags, long double out[])```
	 ```void autocorr_fr16 (const fract16 samples[], int sample_length,```
	 ``` int lags, fract16 out[])```
	 ```void autocorr_fr32 (const fract32 samples[], int sample_length,```
	 ``` int lags, fract32 out[])```
	 ```void autocorr_fx16 (const _Fract samples[], int sample_length,```
	 ``` int lags, _Fract out[])```
	 ```void autocorr_fx32 (const long _Fract samples[], int sample_length,```
	 ``` int lags, long _Fract out[])```

Cross-coherence	```void crosscohf (const float samples_a[], const float samples_b[],```
	 ``` int sample_length, int lags, float out[])```
	 ```void crosscoh (const double samples_a[], const double samples_b[],```
	 ``` int sample_length, int lags, double out[])```
	 ```void crosscohd (const long double samples_a[],```
	 ``` const long double samples_b[],```
	 ``` int sample_length, int lags, long double out[])```
	 ```void crosscoh_fr16 (const fract16 samples_a[],```
	 ``` const fract16 samples_b[],```
	 ``` int sample_length, int lags, fract16 out[])```
	 ```void crosscoh_fr32 (const fract32 samples_a[],```
	 ``` const fract32 samples_b[],```
	 ``` int sample_length, int lags, fract32 out[])```
	 ```void crosscoh_fx16 (const _Fract samples_a[], const _Fract samples_b[],```
	 ``` int sample_length, int lags, _Fract out[])```
	 ```void crosscoh_fx32 (const long _Fract samples_a[],```
	 ``` const long _Fract samples_b[],```
	 ``` int sample_length, int lags, long _Fract out[])```

Cross-correlation	```void crosscorrf (const float samples_a[], const float samples_b[],```
	 ``` int sample_length, int lags, float out[])```
	 ```void crosscorr (const double samples_a[], const double samples_b[],```
	 ``` int sample_length, int lags, double out[])```
	 ```void crosscorrd (const long double samples_a[],```
	 ``` const long double samples_b[],```
	 ``` int sample_length, int lags, long double out[])```
	 ```void crosscorr_fr16 (const fract16 samples_a[],```
	 ``` const fract16 samples_b[],```
	 ``` int sample_length, int lags, fract16 out[])```
	 ```void crosscorr_fx16 (const _Fract samples_a[], const _Fract samples_b[],```
	 ``` int sample_length, int lags, _Fract out[])```

Table 5-7: Statistical Functions (Continued)

Description	Prototype
| | ```
void crosscorr_fr32 (const fract32 samples_a[],
                     const fract32 samples_b[],
                     int sample_length, int lags, fract32 out[])
void crosscorr_fx32 (const long _Fract samples_a[],
                     const long _Fract samples_b[],
                     int sample_length, int lags, long _Fract out[])
``` |
| Histogram | ```
void histogramf (const float samples[], int out[], float max_sample,
                 float min_sample, int sample_length, int bin_count)
void histogram (const double samples[], int out[], double max_sample,
                double min_sample, int sample_length, int bin_count)
void histogramd (const long double samples[], int out[],
                 long double max_sample,
                 long double min_sample, int sample_length,
                 int bin_count)
void histogram_fr16 (const fract16 samples[], int out[],
                     fract16 max_sample,
                     fract16 min_sample, int sample_length,
                     int bin_count)
void histogram_fx16 (const _Fract samples[], int out[],
                     _Fract max_sample,
                     _Fract min_sample, int sample_length,
                      int bin_count)
void histogram_fr32 (const fract32 samples[], int out[],
                     fract32 max_sample,
                     fract32 min_sample, int sample_length,
                     int bin_count)
void histogram_fx32 (const long _Fract samples[], int out[],
                     long _Fract max_sample,
                     long _Fract min_sample, int sample_length,
                     int bin_count)
``` |
| Mean | ```
float meanf (const float samples[], int sample_length)
double mean (const double samples[], int sample_length)
long double meand (const long double samples[], int sample_length)
fract16 mean_fr16 (const fract16 samples[], int sample_length)
_Fract mean_fx16 (const _Fract samples[], int sample_length)
fract32 mean_fr32 (const fract32 samples[], int sample_length)
long _Fract mean_fx32 (const long _Fract samples[], int sample_length)
``` |
| Root Mean Square | ```
float rmsf (const float samples[], int sample_length)
double rms (const double samples[], int sample_length)
long double rmsd (const long double samples[], int sample_length)
fract16 rms_fr16 (const fract16 samples[], int sample_length)
fract32 rms_fr32 (const fract32 samples[], int sample_length)
_Fract rms_fx16 (const _Fract samples[], int sample_length)
long _Fract rms_fx32 (const long _Fract samples[], int sample_length)
``` |
| Variance | ```
float varf (const float samples[], int sample_length)
double var (const double samples[], int sample_length)
long double vard (const long double samples[], int sample_length)
fract16 var_fr16 (const fract16 samples[], int sample_length)
_Fract var_fx16 (const _Fract samples[], int sample_length)
``` |

Table 5-7: Statistical Functions (Continued)

Description	Prototype
	`fract32 var_fr32 (const fract32 samples[], int sample_length)` `long _Fract var_fx32 (const long _Fract samples[], int sample_length)`

Count Zero Crossing	`int zero_crossf (const float samples[], int sample_length)`
	`int zero_cross (const double samples[], int sample_length)`
	`int zero_crossd (const long double samples[], int sample_length)`
	`int zero_cross_fr16 (const fract16 samples[], int sample_length)`
	`int zero_cross_fx16 (const _Fract samples[], int sample_length)`
	`int zero_cross_fr32 (const fract32 samples[], int sample_length)`
	`int zero_cross_fx32 (const long _Fract samples[], int sample_length)`

vector.h

The `vector.h` header file contains functions for operating on real and complex vectors, both vector-scalar and vector-vector operations. See complex.h for definitions of the complex types.

The functions defined in the `vector.h` header file are listed in the *Vector Functions* table. Vector functions that operate on the `complex_fract16` and `complex_fract32` data types, and on the Embedded C data types `_Fract` and `long _Fract`, use saturating arithmetic.

In the *Prototype* column, `vec[]`, `vec_a[]`, and `vec_b[]` are input vectors, `scalar` is an input scalar, `out[]` is an output vector, and `sample_length` is the number of elements. The functions assume that input array arguments are constant; that is, their contents will not change during the course of the routine. In particular, this means the input arguments do not overlap with any output argument. In general, better run-time performance is achieved by the vector functions when the input vectors and the output vector are in different memory banks. This structure avoids any potential memory bank collisions.

Table 5-8: Vector Functions

Description	Prototype

Real Vector + Scalar Addition	`void vecsadd (const double vec[], double scalar,`
	`             double out[], int length)`
	`void vecsaddd (const long double vec[], long double scalar,`
	`             long double out[], int length)`
	`void vecsaddf (const float vec[], float scalar,`
	`             float out[], int length)`
	`void vecsadd_fr16 (const fract16 vec[], fract16 scalar,`
	`               fract16 out[], int length)`
	`void vecsadd_fx16 (const _Fract vec[], _Fract scalar,`
	`               _Fract out[], int length)`
	`void vecsadd_fr32 (const fract32 vec[], fract32 scalar,`
	`               fract32 out[], int length)`
	`void vecsadd_fx32 (const long _Fract vec[],`
	`               long _Fract scalar,`
	`               long _Fract out[], int length)`

Table 5-8: Vector Functions (Continued)

Description	*Prototype*
| Real Vector - Scalar Subtraction | ```
void vecssub (const double vec[], double scalar,
              double out[], int length)
void vecssubd (const long double vec[], long double scalar,
               long double out[], int length)
void vecssubf (const float vec[], float scalar,
               float out[], int length)
void vecssub_fr16 (const fract16 vec[], fract16 scalar,
                   fract16 out[], int length)
void vecssub_fx16 (const _Fract vec[], _Fract scalar,
                   _Fract out[], int length)
void vecssub_fr32 (const fract32 vec[], fract32 scalar,
                   fract32 out[], int length)
void vecssub_fx32 (const long _Fract vec[],
                   long _Fract scalar,
                   long _Fract out[], int length)
``` |
| Real Vector * Scalar Multiplication | ```
void vecsmlt (const double vec[], double scalar,
              double out[], int length)
void vecsmltd (const long double vec[], long double scalar,
               long double out[], int length)
void vecsmltf (const float vec[], float scalar,
               float out[], int length)
void vecsmlt_fr16 (const fract16 vec[], fract16 scalar,
                   fract16 out[], int length)
void vecsmlt_fx16 (const _Fract vec[], _Fract scalar,
                   _Fract out[], int length)
void vecsmlt_fr32 (const fract32 vec[], fract32 scalar,
                   fract32 out[], int length)
void vecsmlt_fx32 (const long _Fract vec[],
                   long _Fract scalar,
                   long _Fract out[], int length)
``` |
| Real Vector + Vector Addition | ```
void vecvadd (const double vec_a[], const double vec_b[],
              double out[], int length)
void vecvaddd (const long double vec_a[],
               const long double vec_b[],
               long double out[], int length)
void vecvaddf (const float vec_a[], const float vec_b[],
               float out[], int length)
void vecvadd_fr16 (const fract16 vec_a[],
                   const fract16 vec_b[],
                   fract16 out[], int length)
void vecvadd_fx16 (const _Fract vec_a[],
                   const _Fract vec_b[],
                   _Fract out[], int length)
void vecvadd_fr32 (const fract32 vec_a[],
                   const fract32 vec_b[],
                   fract32 out[], int length)
void vecvadd_fx32 (const long _Fract vec_a[],
                   const long _Fract vec_b[],
                   long _Fract out[], int length)
``` |

Table 5-8: Vector Functions (Continued)

Description	Prototype

Real Vector - Vector Subtraction	```
	void vecvsub (const double vec_a[], const double vec_b[ ],
	            double out[], int length)
	void vecvsubd (const long double vec_a[],
	            const long double vec_b[],
	            long double out[], int length)
	void vecvsubf (const float vec_a[], const float vec_b[],
	            float out[], int length)
	void vecvsub_fr16 (const fract16 vec_a[],
	            const fract16 vec_b[],
	            fract16 out[], int length)
	void vecvsub_fx16 (const _Fract vec_a[], const _Fract vec_b[],
	            _Fract out[], int length)
	void vecvsub_fr32 (const fract32 vec_a[],
	            const fract32 vec_b[],
	            fract32 out[], int length)
	void vecvsub_fx32 (const long _Fract vec_a[],
	            const long _Fract vec_b[],
	            long _Fract out[], int length)
	``

Real Vector * Vector Multiplication	```
	void vecvmlt (const double vec_a[],const double vec_b[],
	            double out[],int length)
	void vecvmltd (const long double vec_a[],
	            const long double vec_b[],
	            long double out[], int length)
	void vecvmltf (const float vec_a[],
	            const float vec_b[],
	            float out[], int length)
	void vecvmlt_fr16 (const fract16 vec_a[],
	            const fract16 vec_b[],
	            fract16 out[], int length)
	void vecvmlt_fx16 (const _Fract vec_a[],
	            const _Fract vec_b[],
	            _Fract out[], int length)
	void vecvmlt_fr32 (const fract32 vec_a[],
	            const fract32 vec_b[],
	            fract32 out[], int length)
	void vecvmlt_fx32 (const long _Fract vec_a[],
	            const long _Fract vec_b[],
	            long _Fract out[], int length)
	``

Maximum Value of Vector Elements	``` double vecmax (const double vec[], int length) long double vecmaxd (const long double vec[], int length) float vecmaxf (const float vec[], int length) fract16 vecmax_fr16 (const fract16 vec[], int length) _Fract vecmax_fx16 (const _Fract vec[], int length) fract32 vecmax_fr32 (const fract32 vec[], int length) long _Fract vecmax_fx32 (const long _Fract vec[], int length) ```
Minimum Value of Vector Elements	``` double vecmin (const double vec[], int length) long double vecmind (const long double vec[], int length) float vecminf (const float vec[], int length) fract16 vecmin_fr16(const fract16 vec[], int length) _Fract vecmin_fx16(const _Fract vec[], int length) ```

Table 5-8: Vector Functions (Continued)

Description	Prototype
	`fract32 vecmin_fr32(const fract32 vec[], int length)` `long _Fract vecmin_fx32 (const long _Fract vec[], int length)`

Index of Maximum Value of Vector Elements	`int vecmaxloc (const double vec[], int length)`
	`int vecmaxlocd (const long double vec[], int length)`
	`int vecmaxlocf(const float vec[], int length)`
	`int vecmaxloc_fr16 (const fract16 vec[], int length)`
	`int vecmaxloc_fx16 (const _Fract vec[], int length)`
	`int vecmaxloc_fr32 (const fract32 vec[], int length)`
	`int vecmaxloc_fx32 (const long _Fract vec[], int length)`

Index of Minimum Value of Vector Elements	`int vecminloc (const double vec[], int length)`
	`int vecminlocd(const long double vec[], int length)`
	`int vecminlocf (const float vec[], int length)`
	`int vecminloc_fr16(const fract16 vec[], int length)`
	`int vecminloc_fx16(const _Fract vec[], int length)`
	`int vecminloc_fr32(const fract32 vec[], int length)`
	`int vecminloc_fx32 (const long _Fract vec[], int length)`

Complex Vector + Scalar Addition	`void cvecsadd (const complex_double vec[],`
	`               complex_double scalar,`
	`               complex_double out[], int length)`
	`void cvecsaddd (const complex_long_double vec[],`
	`                complex_long_double scalar,`
	`                complex_long_double out[], int length)`
	`void cvecsaddf (const complex_float vec[],`
	`                complex_float scalar,`
	`                complex_float out[], int length)`
	`void cvecsadd_fr16 (const complex_fract16 vec[],`
	`                    complex_fract16 scalar,`
	`                    complex_fract16 out[], int length)`
	`void cvecsadd_fr32 (const complex_fract32 vec[],`
	`                    complex_fract32 scalar,`
	`                    complex_fract32 out[], int length)`

Complex Vector - Scalar Subtraction	`void cvecssub (const complex_double vec[],`
	`               complex_double scalar,`
	`               complex_double out[], int length)`
	`void cvecssubd (const complex_long_double vec[],`
	`                complex_long_double scalar,`
	`                complex_long_double out[], int length)`
	`void cvecssubf (const complex_float vec[],`
	`                complex_float scalar,`
	`                complex_float out[],  int length)`
	`void cvecssub_fr16 (const complex_fract16 vec[],`
	`                    complex_fract16 scalar,`
	`                    complex_fract16 out[], int length)`
	`void cvecssub_fr32 (const complex_fract32 vec[],`
	`                    complex_fract32 scalar,`
	`                    complex_fract32 out[], int length)`

Complex Vector * Scalar Multiplication	`void cvecsmlt (const complex_double vec[],` `               complex_double scalar,`

Table 5-8: Vector Functions (Continued)

Description	Prototype
| | ```
                 complex_double out[], int length)
void cvecsmltd (const complex_long_double vec[],
                complex_long_double scalar,
                complex_long_double out[], int length)
void cvecsmltf (const complex_float vec[],
                complex_float scalar,
                complex_float out[], int length)
void cvecsmlt_fr16 (const complex_fract16 vec[],
                    complex_fract16 scalar,
                    complex_fract16 out[], int length)
void cvecsmlt_fr32 (const complex_fract32 vec[],
                    complex_fract32 scalar,
                    complex_fract32 out[], int length)
``` |
| Complex Vector + Vector Addition | ```
void cvecvadd (const complex_double vec_a[],
               const complex_double vec_b[],
               complex_double out[], int length)
void cvecvaddd (const complex_long_double vec_a[],
                const complex_long_double vec_b[],
                complex_long_double out[],  int length)
void cvecvaddf (const complex_float vec_a[],
                const complex_float vec_b[],
                complex_float out[],    int length)
void cvecvadd_fr16 (const complex_fract16 vec_a[],
                    const complex_fract16 vec_b[],
                    complex_fract16 out[], int length)
void cvecvadd_fr32 (const complex_fract32 vec_a[],
                    const complex_fract32 vec_b[],
                    complex_fract32 out[], int length)
``` |
| Complex Vector - Vector Subtraction | ```
void cvecvsub (const complex_double vec_a[],
               const complex_double vec_b[ ],
               complex_double out[], int length)
void cvecvsubd (const complex_long_double vec_a[],
                const complex_long_double vec_b[],
                complex_long_double out[], int length)
void cvecvsubf (const complex_float vec_a[],
                const complex_float vec_b[],
                complex_float out[], int length)
void cvecvsub_fr16 (const complex_fract16 vec_a[],
                    const complex_fract16 vec_b[],
                    complex_fract16 out[], int length)
void cvecvsub_fr32 (const complex_fract32 vec_a[],
                    const complex_fract32 vec_b[],
                    complex_fract32 out[], int length)
``` |
| Complex Vector * Vector Multiplication | ```
void cvecvmlt (const complex_double vec_a[],
               const complex_double vec_b[],
               complex_double out[], int length)
void cvecvmltd (const complex_long_double vec_a[],
                const complex_long_double vec_b[],
                complex_long_double out[], int length)
void cvecvmltf (const complex_float vec_a[],
``` |

Table 5-8: Vector Functions (Continued)

Description	Prototype
	```const complex_float vec_b[],``` ```complex_float out[], int length)``` ```void cvecvmlt_fr16 (const complex_fract16 vec_a[],``` ```const complex_fract16 vec_b[],``` ```complex_fract16 out[], int length)``` ```void cvecvmlt_fr32 (const complex_fract32 vec_a[],``` ```const complex_fract32 vec_b[],``` ```complex_fract32 out[], int length)```
Real Vector Dot Product	```double vecdot (const double vec_a[],``` ```const double vec_b[], int length)``` ```long double vecdotd (const long double vec_a[],``` ```const long double vec_b[], int length)``` ```float vecdotf (const float vec_a[],``` ```const float vec_b[], int length)``` ```fract16 vecdot_fr16 (const fract16 vec_a[],``` ```const fract16 vec_b[], int length)``` ```_Fract vecdot_fx16 (const _Fract vec_a[],``` ```const _Fract vec_b[], int length)``` ```fract32 vecdot_fr32 (const fract32 vec_a[],``` ```const fract32 vec_b[], int length)``` ```long _Fract vecdot_fx32 (const long _Fract vec_a[],``` ```const long _Fract vec_b[],``` ```int length)```
Complex Vector Dot Product	```complex_double cvecdot (const complex_double vec_a[],``` ```const complex_double vec_b[],``` ```int length)``` ```complex_long_double cvecdotd (``` ```const complex_long_double vec_a[],``` ```const complex_long_double vec_b[],``` ```int length)``` ```complex_float cvecdotf (const complex_float vec_a[],``` ```const complex_float vec_b[],``` ```int length)``` ```complex_fract16 cvecdot_fr16 (const complex_fract16 vec_a[],``` ```const complex_fract16 vec_b[],``` ```int length)``` ```complex_fract32 cvecdot_fr32 (const complex_fract32 vec_a[],``` ```const complex_fract32 vec_b[],``` ```int length)```

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 and described in DSP Run-Time Library Reference.

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

Table 5-9: Window Generator Functions

Description	Prototype
Generate Bartlett Window	```void gen_bartlett_fr16 (fract16 bartlett_window[], int window_stride, int window_size) void gen_bartlett_fx16 (_Fract bartlett_window[], int window_stride, int window_size) void gen_bartlett_fr32 (fract32 bartlett_window[], int window_stride, int window_size) void gen_bartlett_fx32 (long _Fract bartlett_window[], int window_stride, int window_size)```
Generate Blackman Window	```void gen_blackman_fr16 (fract16 blackman_window[], int window_stride, int window_size) void gen_blackman_fx16 (_Fract blackman_window[], int window_stride, int window_size) void gen_blackman_fr32 (fract32 blackman_window[], int window_stride, int window_size) void gen_blackman_fx32 (long _Fract blackman_window[], int window_stride, int window_size)```
Generate Gaussian Window	```void gen_gaussian_fr16 (fract16 gaussian_window[], float alpha, int window_stride, int window_size) void gen_gaussian_fx16 (_Fract gaussian_window[], float alpha, int window_stride, int window_size) void gen_gaussian_fr32 (fract32 gaussian_window[], long double alpha, int window_stride, int window_size) void gen_gaussian_fx32 (long _Fract gaussian_window[], long double alpha, int window_stride, int window_size)```
Generate Hamming Window	```void gen_hamming_fr16 (fract16 hamming_window[], int window_stride, int window_size) void gen_hamming_fx16 (_Fract hamming_window[], int window_stride, int window_size) void gen_hamming_fr32 (fract32 hamming_window[], int window_stride, int window_size) void gen_hamming_fx32 (long _Fract hamming_window[], int window_stride, int window_size)```
Generate Hanning Window	```void gen_hanning_fr16 (fract16 hanning_window[], int window_stride, int window_size) void gen_hanning_fx16 (_Fract hanning_window[], int window_stride, int window_size) void gen_hanning_fr32 (fract32 hanning_window[], int window_stride, int window_size)```

Table 5-9: Window Generator Functions (Continued)

Description	Prototype
	```void gen_hanning_fx32 (long _Fract hanning_window[], int window_stride,``` ``` int window_size)```

Generate Harris Window	```void gen_harris_fr16 (fract16 harris_window[], int window_stride,```
	``` int window_size)```
	```void gen_harris_fx16 (_Fract harris_window[], int window_stride,```
	``` int window_size)```
	```void gen_harris_fr32 (fract32 harris_window[], int window_stride,```
	``` int window_size)```
	```void gen_harris_fx32 (long _Fract harris_window[], int window_stride,```
	``` int window_size)```

Generate Kaiser Window	```void gen_kaiser_fr16 (fract16 kaiser_window[], float beta,```
	``` int window_stride, int window_size)```
	```void gen_kaiser_fx16 (_Fract kaiser_window[], float beta,```
	``` int window_stride, int window_size)```
	```void gen_kaiser_fr32 (fract32 kaiser_window[], long double beta,```
	``` int window_stride, int window_size)```
	```void gen_kaiser_fx32 (long _Fract kaiser_window[], long double beta,```
	``` int window_stride, int window_size)```

Generate rectangular Window	```void gen_rectangular_fr16 (fract16 rectangular_window[],```
	``` int window_stride, int window_size)```
	```void gen_rectangular_fx16 (_Fract rectangular_window[],```
	``` int window_stride, int window_size)```
	```void gen_rectangular_fr32 (fract32 rectangular_window[],```
	``` int window_stride, int window_size)```
	```void gen_rectangular_fx32 (long _Fract rectangular_window[],```
	``` int window_stride, int window_size)```

Generate triangle Window	```void gen_triangle_fr16 (fract16 triangle_window[],```
	``` int window_stride, int window_size)```
	```void gen_triangle_fx16 (_Fract triangle_window[],```
	``` int window_stride, int window_size)```
	```void gen_triangle_fr32 (fract32 triangle_window[],```
	``` int window_stride, int window_size)```
	```void gen_triangle_fx32 (long _Fract triangle_window[],```
	``` int window_stride, int window_size)```

Generate von Hann Window	```void gen_vonhann_fr16 (fract16 vonhann_window[],```
	``` int window_stride, int window_size)```
	```void gen_vonhann_fx16 (_Fract vonhann_window[],```
	``` int window_stride, int window_size)```
	```void gen_vonhann_fr32 (fract32 vonhann_window[],```
	``` int window_stride, int window_size)```
	```void gen_vonhann_fx32 (long _Fract vonhann_window[],```
	``` int window_stride, int window_size)```

Measuring Cycle Counts

The common basis for benchmarking some arbitrary C-written source is to measure the number of processor cycles that the code uses. Once known, calculate the actual time taken by multiplying the number of processor cycles by the clock rate of the processor.

NOTE: The cycle counting macros detailed in this section are not thread-safe. If the cycle counting macros are to be used in a multi-threaded environment, they should be invoked from a critical region.

The run-time library provides three alternative methods for measuring processor counts, as described in the following sections:

- 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 to read the register again after the code has been executed. This process is represented by two macros 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 is then passed to the macro `STOP_CYCLE_COUNT`, which calculates 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 parameter `T`) will be adjusted to allow for this additional cost. Parameters `S` and `T` must be separate variables; they should be declared as a `cycle_t` data type, which the header file `cycle_count.h` defines as:

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

NOTE: The use of the volatile type qualifier in the definition of the `cycle_t` data type means that `cycle_t` cannot be specified as a function return type.

The header file also defines the `PRINT_CYCLES(STRING,T)` macro, which is provided mainly as an example of how to print a value of type `cycle_t`; the macro outputs the text `STRING` to `stdout` followed by the number of cycles `T`.

The instrumentation represented by the macros defined in this section is activated only when the program is compiled with the `-DDO_CYCLE_COUNTS` compile-time switch. If this switch is not specified, 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; it may therefore be customized for a particular application (if required), without having to rebuild 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 cycle-count registers of the Blackfin architecture, the macros can also accumulate statistics suited to recording the performance of a section of code that is executed repeatedly.

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

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

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

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

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

- CYCLES_RESET(S) - This macro re-zeros the accumulated statistics 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 -DDO_CYCLE_COUNTS switch is not specified, the macros described above are defined as null macros and no cycle-count information is gathered. To switch between development and release mode therefore requires re-compilation and does not require any changes to the source of an application.

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

The following example demonstrates how this facility may be used.

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

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

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

   CYCLES_INIT(stats);

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

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

This example may 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
```

Alternative methods of measuring the performance of compiled C source are described in Basic Cycle-Counting Facility and Using time.h to Measure Cycle Counts. Also refer to Considerations When Measuring Cycle Counts, which provides useful tips with regards to performance measurements.

Using time.h to Measure Cycle Counts

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

In the ANSI C standard, the clock function is defined to return the number of implementation-dependent clock "ticks" that have elapsed since the program began. In this version of the C/C++ compiler, the clock 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 verified independently 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) (stop_time - start_time))
           / 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 Basic Cycle-Counting Facility and Cycle-Counting Facility With Statistics, respectively. Also refer to Considerations When Measuring Cycle Counts, which provides useful guidelines.

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

On Blackfin processors, it is set by the run-time library to one of the following values in descending order of precedence:

- By way of the `-DCLOCKS_PER_SEC=<definition>` compile-time switch. Because the `time_t` type is based on the `long long int` data type, it is recommended that the value assigned to the symbolic name `CLOCKS_PER_SEC` is defined as the same data type by qualifying the value with the `LL` (or `ll`) suffix (for example, `-DCLOCKS_PER_SEC=60000000LL`).

- By way of the System Services Library

- By way of the *Processor speed* option, found at *Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Processor*

- From the `cycles.h` header file

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

Considerations When Measuring Cycle Counts

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

- Basic Cycle-Counting Facility. This 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 on or off, 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 offers more features than the basic cycle-counting facility described above. It is more expensive in terms of program memory, data memory, and cycles consumed. However, it can record the number of times that the instrumented code has been executed and can calculate the maximum, minimum, and average cost of each iteration. The provided macros take into account the overhead involved in reading the cycle-count register. By default, the macros are switched off, but they can be switched on by specifying the `-DDO_CYCLE_COUNTS` compile-time switch. These macros may also be customized for a specific application. This cycle-counting facility is 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 small sections of code that run for a much shorter time span.

  When benchmarking code, some thought is required when adding timing instrumentation to C source that will be optimized. If the sequence of statements to be measured is not selected carefully, the optimizer may move instructions into (and out of) the code region and/or it may re-site the instrumentation itself, 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 be simple scalars that are allocated in internal memory (be they assigned the result of a reference to the `clock` function, or be they used as arguments to the cycle-counting macros). In the case of variables that are assigned the result of the `clock` function, it is also recommended that they be defined with the `volatile` keyword.

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

  NOTE: The cycle counting macros detailed in this section are not thread-safe because a context switch may occur between the reading of the `CYCLES` and `CYCLES2` registers. If the cycle counting macros are to be used in a multi-threaded environment, they should be invoked from a critical region.

DSP Run-Time Library Reference

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

Notation Conventions

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

Reference Format

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

- *Name* - 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

a_compress

A-law compression

Synopsis

```
#include <filter.h>

void a_compress(const short input[], short output[], int length);
```

Description

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

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

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

Domain

Content of input array: `[-4096, 4095]`

a_expand

A-law expansion

Synopsis

```
#include <filter.h>

void a_expand (const short input[], short output[], int length);
```

Description

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

> **NOTE:** This function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

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

Domain

Content of input array: `[0, 255]`

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.

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

$c = e^x$

Domain

x = [-87.33, 88.72]	for alogf( )
x = [-708.39, 709.78]	for alogd( )

Example

```
#include <math.h>

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

See Also

alog10, exp, log, pow

alog10

Base 10 anti-log

Synopsis

```
#include <math.h>

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

Description

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

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

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

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

Domain

x = [-37.92 , 38.53]	for `alog10f( )`
x = [-307.65 , 308.25]	for `alog10d( )`

Example

```
#include <math.h>

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

See Also

alog, exp, log, pow

arg

Get phase of a complex number

Synopsis

```
#include <complex.h>

float argf (complex_float a);
double arg (complex_double a);
long double argd (complex_long_double a);

fract16 arg_fr16 (complex_fract16 a);
fract32 arg_fr32 (complex_fract32 a);
_Fract arg_fx_fr16 (complex_fract16 a);
long _Fract arg_fx_fr32 (complex_fract32 a);
```

Description

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

NOTE: Refer to the description of the polar function, which explains how a phase, represented as a fractional number, is interpreted in polar notation.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equation is the basis of the algorithm.

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

Domain

[-3.4e38 , +3.4e38]	for `argf( )`
[-1.7 e308 , +1.7e308]	for `argd( )`
[-1.0 , +1.0)	for `arg_fr16( )`, `arg_fx_fr16( )`, `arg_fr32( )`, `arg_fx_fr32( )`

Note

Im (a) / Re (a) <=1	for `arg_fr16( )`, `arg_fx_fr16( )`

autocoh

Auto-coherence

Synopsis

```
#include <stats.h>

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

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

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

void autocoh_fr16 (const fract16   samples[ ],
```

```
                    int      sample_length,
                    int      lags,
                    fract16  coherence[ ]);

void autocoh_fr32 (const fract32  samples[ ],
                    int      sample_length,
                    int      lags,
                    fract32  coherence[ ]);

void autocoh_fx16 (const _Fract   samples[ ],
                     int      sample_length,
                     int      lags,
                    _Fract   coherence[ ]);

void autocoh_fx32 (const long _Fract   samples[ ],
                    int          sample_length,
                    int          lags,
                    long _Fract   coherence[ ]);
```

Description

The `autocoh` functions compute the auto-coherence of the signal contained in `samples`, of length
`sample_length`. 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 `coherence` of length `lags`.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2
> Utility ROM for further information.

Error Conditions

The `autocoh` functions 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.

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

where:

n = `sample_length`

k = { 0, 1, ..., `lags-1` }

CCES 2.9.0 C/C++ Compiler and Library Manual for Blackfin Processors

a = samples

Domain

[-3.4e38 , +3.4e38]	for autocohf( )
[-1.7e308 , +1.7e308]	for autocohd( )
[-1.0 , 1.0)	for autocoh_fr16( ), autocoh_fx16( ), autocoh_fr32( ), autocoh_fx32( )

Example

```
#include <stats.h>

#define SAMPLES 1024
#define LAGS       16

fract32 x [SAMPLES];fract32 response[LAGS];
autocoh_fr32 (x, SAMPLES, LAGS, response);
```

See Also

autocorr, crosscoh, crosscorr

autocorr

Autocorrelation

Synopsis

```
#include <stats.h>

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

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

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

void autocorr_fr16 (const fract16   samples[ ],
```

```
                            int      sample_length,
                            int      lags,
                            fract16  correlation[ ]);

void autocorr_fr32 (const fract32  samples[ ],
                            int      sample_length,
                            int      lags,
                            fract32  correlation[ ]);

void autocorr_fx16 (const _Fract    samples[ ],
                             int      sample_length,
                             int      lags,
                            _Fract    correlation[ ]);

void autocorr_fx32 (const long _Fract    samples[ ],
                            int            sample_length,
                            int            lags,
                            long _Fract    correlation[ ]);
```

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 `samples[]` input array. The number of samples of the autocorrelation sequence to be produced is given by `lags`. The length of the input sequence is given by `sample_length`.

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equation is the basis of the algorithm.

$$c_k = \frac{1}{n} \sum_{j=0}^{n-k-1} a_j \cdot a_{j+k}$$

where:

a = `samples`;

k = {0, 1, ..., m-1}

m is the number of `lags`

n is the *size* of the input vector `samples`

Domain

[-3.4e38 , +3.4e38]	for `autocorrf( )`
[-1.7e308 , +1.7e308]	for `autocorrd( )`
[-1.0 , 1.0)	for `autocorr_fr16( )` and `autocorr_fx16( )`
	for `autocorr_fr32( )` and `autocorr_fx32( )`

cabs

Complex absolute value

Synopsis

```
#include <complex.h>

float cabsf (complex float a);
double cabs (complex double a);
long double cabsl (complex long double a);

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

fract16 cabs_fr16 (complex_fract16 a);
fract32 cabs_fr32 (complex_fract32 a);
_Fract cabs_fx_fr16 (complex_fract16 a);
long _Fract cabs_fx_fr32 (complex_fract32 a);
```

Description

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

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.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for more information.

Algorithm

The following equation is the basis of the algorithm.

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

Domain

Re^2 (a) + Im^2 (a) <= 3.4 x 10^{38}	for `cabsf( )`
Re^2 (a) + Im^2 (a) <= 1.7 x 10^{308}	for `cabsd( )`
Re^2 (a) + Im^2 (a) <= 1.0	for `cabs_fr16( )` and `cabs_fx_fr16( )`
	for `cabs_fr32( )` and `cabs_fx_fr32( )`

cadd

Complex addition

Synopsis

```
#include <complex.h>

complex_float caddf (complex_float a, complex_float b);
complex_double cadd (complex_double a, complex_double b);
complex_long_double caddd (complex_long_double a, complex_long_double b);

complex_fract16 cadd_fr16 (complex_fract16 a, complex_fract16 b);
complex_fract32 cadd_fr32 (complex_fract32 a, complex_fract32 b);
```

Description

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

> **NOTE:** The `caddd` function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

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

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

Domain

[-3.4e38 , +3.4e38]	for `caddf( )`
[-1.7e308 , +1.7e308]	for `caddd( )`

[-1.0 , +1.0)	for `cadd_fr16( )`
	for `cadd_fr32( )`

cartesian

Convert Cartesian to polar notation

Synopsis

```
#include <complex.h>

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

fract16 cartesian_fr16 (complex_fract16 a, fract16 *phase);
fract32 cartesian_fr32 (complex_fract32 a, fract32 *phase);
_Fract cartesian_fx_fr16 (complex_fract16 a, _Fract *phase);
long _Fract cartesian_fx_fr32 (complex_fract32 a, long _Fract *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`.

> **NOTE:** Refer to the polar function, which explains how a phase, represented as a fractional number, is interpreted in polar notation.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

magnitude = cabs(a)

phase = arg(a)

Domain

[-3.4e38 , +3.4e38]	for `cartesianf( )`
[-1.7e308 , +1.7e308]	for `cartesiand( )`
[-1.0 , +1.0)	for `cartesian_fr16( )` and `cartesian_fx_fr16( )`
	for `cartesian_fr32( )` and `cartesian_fx_fr32( )`

Example

```
#include <complex.h>

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

cdiv

Complex division

Synopsis

```
#include <complex.h>

complex_float cdivf (complex_float  a, complex_float b);
complex_double cdiv (complex_double a, complex_double b);
complex_long_double cdivd (complex_long_double a, complex_long_double b);

complex_fract16 cdiv_fr16 (complex_fract16 a, complex_fract16 b);
complex_fract32 cdiv_fr32 (complex_fract32 a, complex_fract32 b);
```

Description

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

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equation is the basis of the algorithm.

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

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

Domain

[-3.4e38 , +3.4e38]	for `cdivf( )`

[-1.7e308 , +1.7e308]	for `cdivd( )`
[-1.0 , +1.0)	for `cdiv_fr16( )` and `cdiv_fr32( )`

cexp

Complex exponential

Synopsis

```
#include <complex.h>

complex_float cexpf (float x);
complex_double cexp (double x);
complex_long_double cexpd (long double x);
```

Description

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

The `cexp` and `cexpf` functions use the same name for both C99 native complex types and `complex_float` / `complex_double` types. Although the C99 native complex functions are not yet available in the library, 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 `ADI_COMPLEX_STRUCT_FORM`.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

Re(c) = cos(x)

Im(c) = sin(x)

Domain

x = [-102940 , 102940]	for `cexpf( )`
x = [-8.433e8 , 8.433e8]	for `cexpd( )`

cfft

Complex radix-2 Fast Fourier Transform

Synopsis

```
#include <filter.h>

void cfft_fr16(const complex_fract16 input[],
                     complex_fract16 output[],
               const complex_fract16 twiddle_table[],
                     int              twiddle_stride,
                     int              fft_size,
                     int             *block_exponent,
                     int              scale_method);

void cfft_fr32(const complex_fract32 input[],
                     complex_fract32 output[],
               const complex_fract32 twiddle_table[],
                     int              twiddle_stride,
                     int              fft_size,
                     int             *block_exponent,
                     int              scale_method);
```

Description

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

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `fft_size`/2 twiddle factors. The table is composed of +`cosine` and –`sine` coefficients and may be initialized by using the function `twidfftrad2_fr16` (see twidfftrad2) and `twidfftrad2_fr32` for `cfft_fr32`. For optimal performance, the twiddle table should be allocated in a different memory section than the output array.

The argument `twiddle_stride` should be set to 1 if the twiddle table was originally created for an FFT of size `fft_size`. If the twiddle table was created for a larger FFT of size N*`fft_size` (where N is a power of 2), then `twiddle_stride` should be set to N. This argument therefore provides a way of using a single twiddle table to calculate FFTs of different sizes.

The argument `scale_method` controls how the function will apply scaling while computing a Fourier Transform. The available options are static scaling (dividing the input at any stage by 2), dynamic scaling (dividing the input at any stage by 2 if the largest absolute input value is greater than or equal to 0.25), or no scaling. Note that the number of stages required to compute an FFT is dependent on the size of the FFT and is given by the formula `log2(fft_size)`.

If static scaling is selected, the function will always scale intermediate results, thus preventing overflow. The loss of precision increases in line with `fft_size` and is more pronounced for input signals with a small magnitude (since

the output is scaled by 1/`fft_size`). To select static scaling, set the argument `scale_method` to a value of 1. The block exponent returned will be `log2(fft_size)`.

If dynamic scaling is selected, the function inspects intermediate results and only apply scaling where required to prevent overflow. The loss of precision increases in line with the size of the FFT and is more pronounced for input signals with a large magnitude (since these factors increase the need for scaling). The requirement to inspect intermediate results will have an impact on performance. To select dynamic scaling, set the argument `scale_method` to a value of 2. The block exponent returned will be between 0 and `log2(fft_size)` depending upon the number of times that the function scales each set of intermediate results.

If no scaling is selected, the function will never scale intermediate results. There will be no loss of precision unless overflow occurs and in this case the function will generate saturated results. The likelihood of saturation increases in line with the `fft_size` and is more pronounced for input signals with a large magnitude. To select no scaling, set the argument `scale_method` to 3. The block exponent returned will be 0.

> **NOTE:** Any values for the argument `scale_method` other than 2 or 3 will result in the function performing static scaling.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The `cfft` functions abort if the FFT size is less than 8 or if the twiddle stride is less than 1.

Algorithm

The following equation is the basis of the algorithm.

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

Domain

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

Example

```
#include <filter.h>

#define FFT_SIZE1    32
#define FFT_SIZE2    256
#define TWID_SIZE   (FFT_SIZE2/2)

complex_fract32    in1[FFT_SIZE1], in2[FFT_SIZE2];
complex_fract32    out1[FFT_SIZE1], out2[FFT_SIZE2];
```

```
complex_fract32   twiddle[TWID_SIZE];
int               block_exponent1, block_exponent2;

twidfftrad2_fr32 (twiddle, FFT_SIZE2);

cfft_fr32 (in1, out1, twiddle, (FFT_SIZE2 / FFT_SIZE1),
           FFT_SIZE1, &block_exponent1, 1    /* static scaling */ );

cfft_fr32 (in2, out2, twiddle, 1, FFT_SIZE2,
           &block_exponent2, 2               /* dynamic scaling */ );
```

cfftf

Fast N-point complex input FFT

Synopsis

```
#include <filter.h>

void cfftf_fr16(const complex_fract16 input[],
                      complex_fract16 output[],
                const complex_fract16 twiddle_table[],
                      int             twiddle_stride,
                      int             fft_size);

void cfftf_fr32(const complex_fract32 input[],
                      complex_fract32 output[],
                const complex_fract32 twiddle_table[],
                      int             twiddle_stride,
                      int             fft_size);
```

Description

The `cfftf` functions transform 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. The functions "decimate in frequency" using a mixed-radix algorithm.

The size of the input array `input` and the output array `output` is `fft_size`, where `fft_size` represents the number of points in the FFT.

The number of points in the FFT must be a power of 2 and must be at least 8.

The twiddle table is passed in the argument `twiddle_table`, which must contain at least 3 * `fft_size` / 4 complex twiddle factors. The table should be initialized with complex twiddle factors in which the real coefficients are positive cosine values and the imaginary coefficients are negative sine values. The function `twidfftf_fr16` (see twidfftf) can be used to initialize the array for `cfftf_fr16`, with `twidfftf_fr32` used to initialize the array for `cfftf_fr32`.

If the twiddle table has been generated for an `fft_size` FFT, then the `twiddle_stride` argument should be set 1. On the other hand, if the twiddle table has been generated for an FFT of size x, where `x > fft_size`, then the `twiddle_stride` argument should be set to `x/fft_size`. The `twiddle_stride` argument, therefore, allows the same twiddle table to be used for different sizes of FFT. (The `twiddle_stride` argument cannot be either zero or negative).

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

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

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equation is the basis of the algorithm.

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

The `cfftf` functions use a mixed-radix algorithm (radix-2 and radix-4).

Domain

The number of points in the FFT must be a power of 2 and must be at least 8.

Example

```
#include <filter.h>
#define FFT_SIZE1    32
#define FFT_SIZE2    256
#define TWID_SIZE    ((3 * FFT_SIZE2) / 4)

complex_fract32    in1[FFT_SIZE1],  in2[FFT_SIZE2];
complex_fract32    out1[FFT_SIZE1], out2[FFT_SIZE2];
complex_fract32    twiddle[TWID_SIZE];

twidfftf_fr32 (twiddle, FFT_SIZE2);
cfftf_fr32 (in1, out1, twiddle, FFT_SIZE2/FFT_SIZE1, FFT_SIZE1);
cfftf_fr32 (in2, out2, twiddle, 1, FFT_SIZE2);
```

cfft2d

N x N point 2-D complex input FFT

Synopsis

```
#include <filter.h>

void cfft2d_fr16(const complex_fract16        *input,
                       complex_fract16        *temp,
                       complex_fract16        *output,
                 const complex_fract16         twiddle_table[],
                       int                     twiddle_stride,
                       int                     fft_size,
                       int                     block_exponent,
                       int                     scale_method);

void cfft2d_fr32(const complex_fract32        *input,
                       complex_fract32        *temp,
                       complex_fract32        *output,
                 const complex_fract32         twiddle_table[],
                       int                     twiddle_stride,
                       int                     fft_size);
```

Description

These `cfft2d` functions compute the two-dimensional Fast Fourier Transform (FFT) of the complex input matrix `input[fft_size][fft_size]` and store the result to the complex output matrix `output[fft_size][fft_size]`.

The size of the input array `input`, the output array `output`, and the temporary working buffer `temp` is `fft_size*fft_size`, where `fft_size` represents the number of rows and number of columns in the FFT. The argument `fft_size` must be a power of 2 and must be at least 4 for `cfft2d_fr16` (at least 8 for `cfft2d_fr32`).

Memory bank collisions, which have an adverse effect on run-time performance, may be avoided by allocating the twiddle table in a different memory bank than the output matrix and temporary buffer. If the input data can be overwritten, the optimum memory usage can be achieved by also specifying the input matrix as the output buffer.

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `fft_size` twiddle factors for `cfft2d_fr16` and at least `3*fft_size/4` twiddle factors for `cfft2d_fr32`. The table should be initialized with complex twiddle factors in which the real coefficients are positive cosine values and the imaginary coefficients are negative sine values. The functions `twidfft2d_fr16` and `twidfft2d_fr32` may be used to initialize the arrays for `cfft2d_fr16` and `cfft2d_fr32` respectively.

If the twiddle table has been generated for an `fft_size` FFT, the `twiddle_stride` argument should be set 1. On the other hand, if the twiddle table has been generated for an FFT of size x, where x>`fft_size`, then the `twiddle_stride` argument should be set to x / `fft_size`. The `twiddle_stride` argument therefore allows the same twiddle table to be used for different sizes of FFT. (The `twiddle_stride` argument cannot be either zero or negative).

To avoid overflow, the functions scale the output by `fft_size*fft_size`.

The `cfft2d_fr16` arguments `block_exponent` and `scale_method` have been added for future expansion. These arguments are ignored by the function.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The `cfft2d` functions abort if the twiddle stride is less than 1, or if `fft_size` is less than 4 for `cfft2d_fr16`, or if `fft_size` is less than 8 for `cfft2d_f32`.

Algorithm

The following equation is the basis of the algorithm.

$$c(i, j) = \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} a(k, l) \cdot e^{(-2\pi \cdot (i \cdot k + j \cdot l))/n}$$

where:

$i = \{0, 1, ..., n-1\}$

$j = \{0, 1, ..., n-1\}$

a = `input`

c = `output`

n = `fft_size`

Domain

Input sequence length `fft_size` must be a power of 2 and at least 4 for `cfft2d_fr16` (at least 8 for `cfft2d_fr32`).

Example

```
#include <filter.h>

#define FFT_SIZE1           128
#define FFT_SIZE2           32
#define TWIDDLE_STRIDE1   (FFT_SIZE1 / FFT_SIZE1)
#define TWIDDLE_STRIDE2   (FFT_SIZE1 / FFT_SIZE2)


complex_fract32   in_a[FFT_SIZE1][FFT_SIZE1]
complex_fract32   in_b[FFT_SIZE2][FFT_SIZE2]
complex_fract32   out[FFT_SIZE2][FFT_SIZE2]
complex_fract32   temp[FFT_SIZE1][FFT_SIZE1]
```

```
complex_fract32  twiddle[(3*FFT_SIZE1)/4]

complex_fract32* in1  = (complex_fract32*)in_a;
complex_fract32* in2  = (complex_fract32*)in_b;
complex_fract32* out2 = (complex_fract32*)out;
complex_fract32* tmp  = (complex_fract32*)temp;

twidfft2d_fr32 (twiddle, FFT_SIZE1);

/* In-place computation */
cfft2d_fr32(in1, tmp, in1, twiddle, TWIDDLE_STRIDE1, FFT_SIZE1);
cfft2d_fr32(in2, tmp, out2, twiddle, TWIDDLE_STRIDE2, FFT_SIZE2);
```

cfir

Complex finite impulse response filter

Synopsis

```
#include <filter.h>

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

void cfir_fr32(const complex_fract32  input[],
                     complex_fract32  output[],
                     int              length,
                     cfir_state_fr32  *filter_state);
```

The `cfir_fr16` function uses the following structure to maintain the state of the filter.

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

The `cfir_fr32` function uses the following structure to maintain the state of the filter.

```
typedef struct { int k;                 /* Number of coefficients */
               complex_fract32 *h;   /* Filter coefficients */
               complex_fract32 *d;   /* Start of delay line */
               complex_fract32 *p; /* Read/write pointer */
} cfir_state_fr32;
```

Description

The `cfir` functions implement a complex finite impulse response (CFIR) filter. They generate the filtered response of the complex input data `input` and store the result in the complex output vector `output`.

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

It is defined as:

```
#define cfir_init(state, coeffs, delay, ncoeffs) \
do {                                             \
    (state).h = (coeffs);   \
    (state).d = (delay);    \
    (state).p = (delay);    \
    (state).k = (ncoeffs);  \
} while (0)
```

The characteristics of the filter (passband, stopband, and so on) depend upon the number of complex filter coefficients and their values. A pointer to the coefficients should be stored in `filter_state->h`, and `filter_state->k` should be set to the number of coefficients. The functions assume that the coefficients are stored in the normal order, thus `filter_state->h[0]` contains the first filter coefficient and `filter_state->h[k-1]` contains the last coefficient.

Each filter should have its own delay line, which is a vector of type `complex_fract16` (for `cfir_fr16`) or `complex_fract32` (for `cfir_fr32`) whose length is equal to the number of coefficients. The vector should be cleared to zero before calling the function for the first time and should not otherwise be modified by the user program. The structure member `filter_state->d` should be set to the start of the delay line, and the function uses `filter_state->p` to keep track of its current position within the vector.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The `cfir` functions check that the number of samples and the number of coefficients are positive - if not, the functions just return.

Algorithm

The following equation is the basis of the algorithm.

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

where:

x = `input`

y = `output`

h = array of coefficients

k = number of coefficients

i = {0, 1, ..., `length` - 1}

Domain

[-1.0 , +1.0)

Example

```
#include <filter.h>

#define LENGTH   85
#define COEFFS_N 32

complex_fract32 input[LENGTH];
complex_fract32 output[LENGTH];
complex_fract32 coeffs[COEFFS_N];
complex_fract32 delay[COEFFS_N];

cfir_state_fr32 state;int i;
for (i=0; i < COEFFS_N; i++)
                        /* clear the delay line */
{  delay[i].re = 0;
   delay[i].im = 0;
}
cfir_init(state, coeffs, delay, COEFFS_N);
cfir_fr32(input, output, LENGTH, &state);
```

clip

Clip

Synopsis

```
#include <math.h>

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

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

```
fract16 clip_fr16 (fract16 parm1, fract16 parm2);
fract32 clip_fr32 (fract32 parm1, fract32 parm2);
_Fract clip_fx16 (_Fract parm1, _Fract parm2);
long _Fract clip_fx32 (long _Fract parm1, long _Fract parm2);
```

Description

The clip 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.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

```
if (|parm1| < |parm2|)
    return (parm1)
else
    return (|parm2| * signof(parm1))
```

Domain

Full range for various input parameter types.

cmlt

Complex multiply

Synopsis

```
#include <complex.h>

complex_float cmltf (complex_float a, complex_float b);
complex_double cmlt (complex_double a, complex_double b);
complex_long_double cmltd (complex_long_double a, complex_long_double b);

complex_fract16 cmlt_fr16 (complex_fract16 a, complex_fract16 b);
complex_fract32 cmlt_fr32 (complex_fract32 a, complex_fract32 b);
```

Description

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

NOTE: The cmltd function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

None

Algorithm

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

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

Domain

[-3.4e38 , +3.4e38]	for `cmltf( )`
[-1.7e308 , +1.7e308]	for `cmltd( )`
[-1.0 , +1.0)	for `cmlt_fr16( )`, `cmlt_fr32( )`

Example

```
#include <complex.h>

complex_fract32 x;
complex_fract32 y;
complex_fract32 z;

z = cmlt_fr32 (x, y);
```

coeff_iirdf1

Convert coefficients for DF1 IIR filter

Synopsis

```
#include <filter.h>

void coeff_iirdf1_fr16 (const float   acoeff[ ],
                        const float   bcoeff[ ],
                        fract16 coeff[ ], int nstages);

void coeff_iirdf1_fx16 (const float acoeff[ ],
                        const float bcoeff[ ],
                        _Fract coeff[ ], int nstages);

void coeff_iirdf1_fr32 (const long double acoeff[ ],
                        const long double bcoeff[ ],
                        fract32 coeff[ ], int nstages);

void coeff_iirdf1_fx32 (const long double acoeff[ ],
```

```
                    const long double bcoeff[ ],
                    long _Fract coeff[ ], int nstages);
```

Description

The `coeff_iirdf1` functions transform a set of A-coefficients and a set of B-coefficients into a set of coefficients for the `iirdf1` functions which implement an optimized, direct form 1 infinite impulse response (IIR) filter. The `coeff_iirdf1_fr16` coefficients are for use with the `iirdf1_fr16` function (see twidfftf), the `coeff_iirdf1_fx16` function coefficients for `iirdf1_fx16`, the `coeff_iirdf1_fr32` function coefficients for `iirdf1_fr32` and the `coeff_iirdf1_fx32` function coefficients are suitable for use with `iirdf1_fx32`.

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

> **NOTE:** For the `coeff_iirdf1_fr16` and `coeff_iirdf1_fx16` functions, the values of the coefficients that are held in the vectors `acoeffs` and `bcoeffs` must be in the range of `[LONG_MIN, LONG_MAX]`; that is, they must not be less than -2147483648, or greater than 2147483647.

The `coeff_iirdf1` functions scale the coefficients and store them in the vector `coeff`. The functions also store the appropriate scaling factor in the vector which the iirdf1 function will then apply to the filtered response that they generate (thus eliminating the need to scale the output generated by the IIR function). The size of `coeffs` array should be `(4*nstages) + 2`.

> **ATTENTION:** Be aware of the consequence of specifying a set of filter coefficients whose order of magnitude are *significantly* different. For instance, when using 16-bit fractional data types, the term "significantly" refers to an order of magnitude greater than or equal to 15 when expressed as a power of 2. In this situation, one or more filter coefficients may be transformed to zero due to the restricted precision of the `fract16` type, and this may affect the performance of the user-designed filter.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

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

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

If any of the coefficients are greater than or equal to 1.0, then all the A-coefficients and all the B-coefficients are scaled to be less than 1.0. The coefficients are stored into the vector `coeffs` in the following order:

$[b_0, -a_01, b_01, -a_02, b_02, ..., -a_n1, b_n1, -a_n2, b_n2, \text{scale factor}]$

where `n` is the number of stages.

NOTE: Note that the A-coefficients are negated by the function.

Domain

The vectors `acoeff` and `bcoeff` must be in the domain `[LONG_MIN, LONG_MAX]` for the `coeff_iirdf1_fr16` and `coeff_iirdf1_fx16` functions, and in the domain `[LLONG_MIN, LLONG_MAX]` for the functions `coeff_iirdf1_fr32` and `coeff_iirdf1_fx32`, where `LONG_MIN`, `LONG_MAX`, `LLONG_MIN` and `LLONG_MAX` are macros that are defined in the `limits.h` header file.

Example

```
#include <filter.h>

#define N_STAGES 25

long double a_coeff[2*N_STAGES];
long double b_coeff[2*N_STAGES+1];
fract32 coefficient[4*N_STAGES+2];

coeff_iirdf1_fr32(a_coeff, b_coeff, coefficient, N_STAGES);
```

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);

complex_fract16 conj_fr16 (complex_fract16 a);
complex_fract32 conj_fr32 (complex_fract32 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.

NOTE: The `conjd` function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

`Re(c) = Re(a)`

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

Domain

[-3.4e38 , +3.4e38]	for `conjf( )`
[-1.7e308 , +1.7e308]	for `conjd( )`
[-1.0 , +1.0)	for `conj_fr16( )`
	for `conj_fr32( )`

convolve

Convolution

Synopsis

```
#include <filter.h>

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

void convolve_fr32(const fract32      input_x[],
                         int           length_x,
                         const fract32  input_y[],
                         int           length_y,
                         fract32        output[]);
```

```
void convolve_fx16(const _Fract        input_x[],
                         int            length_x,
                         const _Fract   input_y[],
                         int            length_y,
                         _Fract         output[]);

void convolve_fx32(const long _Fract      input_x[],
                         int               length_x,
                         const long _Fract input_y[],
                         int               length_y,
                         long _Fract       output[]);
```

Description

The convolution functions convolve two sequences pointed to by input_x and input_y. If input_x points to the sequence whose length is length_x and input_y points to the sequence whose length is length_y, the resulting sequence pointed to by output has length length_x + length_y - 1.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

Convolution between two sequences input_x and input_y is described as:

$$cout[n] = \sum_{k=0}^{clen2-1} cin1[n+k-(clen2-1)] \cdot cin2[(clen2-1)-k]$$

for n = 0 to clen1 + clen2 - 2.

Values for cin1[j] are considered to be zero for j<0 or j>clen1 - 1, where:

cin1 = input_x

cin2 = input_y

cout = output

clen1 = length_x

clen2 = length_y

Domain

[-1.0 , +1.0)

Example

The following is an example of a convolution where `input_x` is of length 4 and `input_y` is of length 3. If we represent `input_x` as "A" and `input_y` as "B", the elements of the output vector are:

```
{A[0]*B[0],
A[1]*B[0] +  A[0]*B[1],
A[2]*B[0] +  A[1]*B[1] + A[0]*B[2],
A[3]*B[0] +  A[2]*B[1] + A[1]*B[2],
A[3]*B[1] + A[2]*B[2],
A[3]*B[2]}
```

conv2d

2-D convolution

Synopsis

```
#include <filter.h>

void conv2d_fr16(const fract16      *input_x,
                       int           rows_x,
                       int           columns_x,
                 const fract16      *input_y,
                       int           rows_y,
                       int           columns_y,
                       fract16      *output);

void conv2d_fx16(const _Fract       *input_x,
                        int          rows_x,
                        int          columns_x,
                 const _Fract       *input_y,
                        int          rows_y,
                        int          columns_y,
                        _Fract       *output);

void conv2d_fr32(const fract32       *input_x,
                       int            rows_x,
                       int            columns_x,
                 const fract32       *input_y,
                       int            rows_y,
                       int            columns_y,
                       fract32       *output);

void conv2d_fx32(const long _Fract   *input_x,
                       int            rows_x,
                       int            columns_x,
                 const long _Fract   *input_y,
                       int             rows_y,
```

```
int            columns_y,
long _Fract    *output);
```

Description

The `conv2d` functions compute the two-dimensional convolution of input matrix `input_x` of size `rows_x*columns_x` and `input_y` of size `rows_y* columns_y` and store the result in matrix `output` of dimension:

`(rows_x + rows_y-1) x (columns_x + columns_y-1)`.

> **NOTE:** A temporary work area is allocated from the run-time stack that the `conv2d_fr16` and `conv2d_fx16` functions use to preserve accuracy while evaluating the algorithm. The stack may therefore overflow if the sizes of the input matrices are sufficiently large. The size of the stack may be adjusted by making appropriate changes to the `.ldf` file.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The `conv2d` functions return if the sizes of any of the dimensions (`rows_x`, `columns_x`, `rows_y`, `columns_y`) are less than or equal to zero.

Algorithm

The two-dimensional convolution of `x[rows_x][cols_x]` and `y[rows_y][cols_y]` is defined as:

$$output[r][c] = \sum_{i=0}^{rows\ x-1} \sum_{k=0}^{cols\ x-1} x[j][k] \cdot y[r-j][c-k]$$

where:

r = 0 to [`rows_x` + `rows_y` - 1]

c = 0 to [`cols_x` + `cols_y` - 1]

Domain

[-1.0 , +1.0)

Example

```
#include <filter.h>

#define ROWS_1 4
#define ROWS_2 4
```

```
#define COLS_1 8
#define COLS_2 2

fract32 input_1[ROWS_1][COLS_1], *a_p = (fract32 *) (&input_1);
fract32 input_2[ROWS_2][COLS_2], *b_p = (fract32 *) (&input_2);
fract32 result[ROWS_1+ROWS_2-1][COLS_1+COLS_2-1];

fract32 *res_p = (fract32 *)(&result);

conv2d_fr32 (a_p, ROWS_1, COLS_1, b_p, ROWS_2, COLS_2, res_p);
```

conv2d3x3

2-D circular convolution with 3 x 3 matrix

Synopsis

```
#include <filter.h>

void conv2d3x3_fr16(const fract16  *input_x,
                    int             rows_x,
                    int             columns_x,
              const fract16  *input_y,
                    fract16  *output);

void conv2d3x3_fx16(const _Fract    *input_x,
                     int             rows_x,
                     int             columns_x,
              const _Fract    *input_y,
                    _Fract    *output);

void conv2d3x3_fr32(const fract32    *input_x,
                    int              rows_x,
                    int              columns_x,
              const fract32    *input_y,
                    fract32    *output);

void conv2d3x3_fx32(const long _Fract *input_x,
                    int              rows_x,
                    int              columns_x,
              const long _Fract *input_y,
                    long _Fract *output);
```

Description

The `conv2d3x3` functions compute the two-dimensional circular convolution of matrix `input_x` with dimensions `[rows_x][columns_x])` and matrix `input_y` with dimensions [3][3], and store the result in matrix `output` with dimensions `[rows_x][columns_x]`.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The `conv2d3x3` functions return if any of the dimensions `rows_x` or `columns_x` are less than or equal to zero.

Algorithm

The two-dimensional circular convolution of `x[rows_x][cols_x]` and `y[3][3]` is defined as:

$$output[r][c] = \sum_{j=0}^{2} \sum_{k=0}^{2} x[(rows\_x-j)\%rows\_x][cols\_x-k)\%cols\_x] \cdot y[j][k]$$

where:

`r` = 0 to `rows_x` - 1

`c` = 0 to `cols_x` - 1

Domain

[-1.0 , +1.0)

Example

```
#include <filter.h>

#define ROWS 9
#define COLS 9

fract32 input_1[ROWS][COLS], *a_p = (fract32 *) (&input_1);
fract32 input_2[3][3],       *b_p = (fract32 *) (&input_2);
fract32 result[ROWS][COLS];

fract32 *res_p = (fract32 *)(&result);

conv2d3x3_fr32 (a_p, ROWS, COLS, b_p, res_p);
```

copysign

Copysign

Synopsis

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

```
float copysignf (float parm1, float parm2);
double copysign (double parm1, double parm2);
long double copysignd (long double parm1, long double parm2);

fract16 copysign_fr16 (fract16 parm1, fract16 parm2);
fract32 copysign_fr32 (fract32 parm1, fract32 parm2);
_Fract copysign_fx16 (_Fract parm1, _Fract parm2);
long _Fract copysign_fx32 (long _Fract parm1, long _Fract parm2);
```

Description

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

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

Domain

Full range for type of parameters used.

cot

Cotangent

Synopsis

```
#include <math.h>

float cotf (float a);
double cot (double a);
long double cotd (long double a);
```

Description

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

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

Domain

a = [-51471, 51471]	for `cotf( )`
a = [-4.21657e8 , 4.21657e8]	for `cotd( )`

countones

Count one bits in word

Synopsis

```
#include <math.h>

int countones(int parm);
int lcountones(long parm);
int llcountones(long long int parm);
```

Description

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equation is the basis of the algorithm.

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

where:

N is the number of bits in `parm`

`bit[j]` represents the jth bit of the parameter `parm`

crosscoh

Cross-coherence

Synopsis

```
#include <stats.h>

void crosscohf (const float   samples_x[ ],
                const float   samples_y[ ],
```

```
                        int    sample_length,
                        int    lags,
                        float  coherence[ ]);

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

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

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

void crosscoh_fr32 (const fract32  samples_x[ ],
                    const fract32  samples_y[ ],
                          int       sample_length,
                          int       lags,
                          fract32  coherence[ ]);

void crosscoh_fx16 (const _Fract   samples_x[ ],
                    const _Fract   samples_y[ ],
                           int      sample_length,
                           int      lags,
                     _Fract   coherence[ ]);

void crosscoh_fx32 (const long _Fract   samples_x[ ],
                    const long _Fract   samples_y[ ],
                          int            sample_length,
                          int            lags,
                          long _Fract   coherence[ ]);
```

Description

The `crosscoh` functions perform a cross-coherence between the two signals contained in `samples_x` and `samples_y`, both of length `samples_length`. 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 `samples_x` and the partial mean of `samples_y`. The cross-coherence between the two input signals is returned in the array `coherence` of length `lags`.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The `crosscoh` functions 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 = `sample_length`

k = { 0, 1, ..., `lags-1` }

a = `samples_x`

b = `samples_y`

Domain

[-3.4e38 , +3.4e38]	for `crosscohf( )`
[-1.7e308 , +1.7e308]	for `crosscohd( )`
[-1.0 , +1.0)	for `crosscoh_fr16( )` and `crosscoh_fx16( )`
	for `crosscoh_fr32( )` and `crosscoh_fx32( )`

Example

```
#include <stats.h>

#define SAMPLES 1024
#define LAGS       16

fract32 x[SAMPLES];
fract32 y[SAMPLES];
fract32 response[LAGS];
crosscoh_fr32 (x, y, SAMPLES, LAGS, response);
```

See Also

autocoh, autocorr, crosscorr

crosscorr

Cross-correlation

Synopsis

```
#include <stats.h>

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

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

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

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

void crosscorr_fx16 (const _Fract  samples_x[ ],
                     const _Fract  samples_y[ ],
                            int     sample_length,
                            int     lags,
                      _Fract        correlation[ ]);

void crosscorr_fr32 (const fract32  samples_x[ ],
                     const fract32  samples_y[ ],
                           int      sample_length,
                           int      lags,
                           fract32  correlation[ ]);

void crosscorr_fx32 (const long _Fract samples_x[ ],
                     const long _Fract samples_y[ ],
```

```
int         sample_length,
int         lags,
long _Fract correlation[ ]);
```

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 vectors `samples_x[]` and `samples_y[]`. The length of the input vectors is given by `sample_length`. The functions return the result in the array `correlation` with `lags` elements.

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equation is the basis of the algorithm.

$$c_k = \frac{1}{n} \cdot \sum_{j=0}^{n-k-1} a_j \cdot b_{j+k}$$

where:

k = {0, 1, ..., `lags`-1}

a = `samples_x`

b = `samples_y`

n = `sample_length`

Domain

[-3.4e38 , +3.4e38]	for `crosscorrf( )`
[-1.7e308 , +1.7e308]	for `crosscorrd( )`
[-1.0 , +1.0)	for `crosscorr_fr16( )`, `crosscorr_fx16( )`, `crosscorr_fr32( )`, `crosscorr_fx32( )`

csub

Complex subtraction

Synopsis

```
#include <complex.h>

complex_float csubf (complex_float a, complex_float b);
complex_double csub (complex_double a, complex_double b);
complex_long_double csubd (complex_long_double a, complex_long_double b);

complex_fract16 csub_fr16 (complex_fract16 a, complex_fract16 b);
complex_fract32 csub_fr32 (complex_fract32 a, complex_fract32 b);
```

Description

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

> **NOTE:** The `csubd` function is included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

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

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

Domain

[-3.4e38 , +3.4e38]	for `csubf( )`
[-1.7e308 , +1.7e308]	for `csubd( )`
[-1.0 , +1.0)	for `csub_fr16( )` and `csub_fr32( )`

fft_magnitude

FFT magnitude

Synopsis

```
#include <filter.h>

void fft_magnitude_fr16(const complex_fract16 input[],
                        fract16 output[], int fft_size,
                        int block_exponent, int mode);
void fft_magnitude_fr32(const complex_fract32 input[],
                        fract32 output[], int fft_size,
                        int block_exponent, int mode);
```

Description

The FFT magnitude functions, `fft_magnitude_fr16` and `fft_magnitude_fr32`, compute a normalized power spectrum from the output signal generated by an FFT function. The `fft_size` argument specifies the size of the FFT and must be a power of 2. The `mode` argument is used to specify the type of FFT function used to generate the input array. The function `fft_magnitude_fr16` computes the magnitude of an FFT that is represented by a `fract16` input array, while `fft_magnitude_fr32` computes the magnitude of an FFT that is represented by a `fract32` input array.

If the input array has been generated from a time-domain complex input signal, the mode argument must be set to 0. Otherwise the `mode` argument must be set to 1 to signify that the input array has been generated from a time-domain real input signal. For example, `mode` must be set to 0 if the input was generated by one of the following library functions:

`cfft_fr16, cfftf_fr16`

`cfft_fr32, cfftf_fr32`

and `mode` must be set to 1 if the input was generated by one of the following library functions:

`rfft_fr16, rfftf_fr16`

`rfft_fr32, rfftf_fr32`

The `block_exponent` argument is used to control the normalization of the power spectrum. It will usually be set to the `block_exponent` that is returned by the `cfft_fr16` or `cfft_fr32`, `rfft_fr16` or `rfft_fr32` functions. If, on the other hand, the input array was generated by one of the functions `cfftf_fr16`, `cfftf_fr32`, `rfftf_fr16` or `rfftf_fr32`, then the `block_exponent` argument should be set to -1, which indicates that the input array was generated using static scaling.

If the input array was generated by some other means, then the value specified for the `block_exponent` argument will depend upon how the FFT was calculated. If the function used to calculate the FFT did not scale the intermediate results at any of the stages of the computation, then set `block_exponent` to zero; if the FFT function scaled the intermediate results at each stage of the computation, then set `block_exponent` to -1; otherwise set `block_exponent` to the number of computation stages that did scale the intermediate results (this value is in the range `0` to `log2(fft_size)`).

> **NOTE:** Functions that compute an FFT using fixed-point arithmetic will usually scale a set of intermediate results to avoid the arithmetic from generating any saturated results. Refer to the description of the `cfft_fr16`, `rfft_fr16` or `cfft_fr32`, `rfft_fr32` functions for more information about different scaling methods.

The `fft_magnitude_fr16` and `fft_magnitude_fr32` functions write the power spectrum to the output array `output`. If `mode` is set to 0, then the length of the power spectrum will be `fft_size`. If `mode` is set to 1, then the length of the power spectrum will be `((fft_size/2)+1)`.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The FFT magnitude functions exit without modifying the output vector if any of the following conditions are true:

- `fft_size` is less than 2

- the `mode` argument is set to a value other than 0 or 1

- `block_exponent` contains a value less than -1

- `block exponent` is greater than 0 and the following condition is not true:

  `fft_size >= (1 << block_exponent)`

Algorithm

For mode 0 (`cfft`-generated input):

$$fft\_magnitude[i] = \frac{sqrt(input[i].re^2 + input[i].im^2)}{fft\_size}$$

where: `i = [0 ... fft_size)`

For mode 1 (`rfft` - generated input):

$$fft\_magnitude[i] = \frac{2 \cdot (sqrt(input[i].re^2 + input[i].im^2))}{fft\_size}$$

where: `i = [0 ... fft_size/2]`

Example

```
#include <filter.h>

#define N_FFT    1024
#pragma align   4096

complex_fract16  cplx_signal[N_FFT];

fract16          real_signal[N_FFT];
complex_fract16  fft_output[N_FFT];
complex_fract16  twiddle_table[N_FFT];

fract16          real_magnitude[(N_FFT/2)+1];
fract16          cplx_magnitude[N_FFT];

int block_exponent;

twidfftrad2_fr16 (twiddle_table, N_FFT);

rfft_fr16 (real_signal, fft_output, twiddle_table, 1, N_FFT, &block_exponent, 2);
```

```
fft_magnitude_fr16 (fft_output, real_magnitude, N_FFT, block_exponent, 1);

twidfftf_fr16 (twiddle_table, N_FFT);

cfftf_fr16 (cplx_signal, fft_output, twiddle_table, 1, N_FFT);

fft_magnitude_fr16 (fft_output, cplx_magnitude, N_FFT, -1, 0);
```

See Also

cfft, cfftf, rfft, rfftf

fir

Finite impulse response filter

Synopsis

```
#include <filter.h>

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

void fir_fx16(const _Fract          input[],
                    _Fract          output[],
                    int             length,
                    fir_state_fx16  *filter_state);

void fir_fr32(const fract32         input[],
                    fract32         output[],
                    int             length,
                    fir_state_fr32  *filter_state);

void fir_fx32(const long _Fract input[],
                    long _Fract     output[],
                    int             length,
                    fir_state_fx32  *filter_state);
```

The `fir_fr16` function uses the following structure to maintain the state of the filter.

```
typedef struct
{
    fract16 *h,   /* filter coefficients */
    fract16 *d,   /* start of delay line */
    fract16 *p,   /* read/write pointer */
    int k;        /* number of coefficients */
```

```
      int l;           /* interpolation/decimation index */
} fir_state_fr16;
```

The `fir_fx16` function uses the following structure to maintain the state of the filter.

```
typedef struct
{
    _Fract *h,    /* filter coefficients */
    _Fract *d,    /* start of delay line */
    _Fract *p,    /* read/write pointer */
     int k;       /* number of coefficients */
     int l;       /* interpolation/decimation index */
} fir_state_fx16;
```

The `fir_fr32` function uses the following structure to maintain the state of the filter.

```
typedef struct
{
    fract32 *h,   /* filter coefficients */
    fract32 *d,   /* start of delay line */
    fract32 *p,   /* read/write pointer */
    int k;        /* number of coefficients */
    int l;        /* interpolation/decimation index */
} fir_state_fr32;
```

The `fir_fx32` function uses the following structure to maintain the state of the filter.

```
typedef struct
{
    long _Fract *h,   /* filter coefficients */
    long _Fract *d,   /* start of delay line */
    long _Fract *p,   /* read/write pointer */
    int k;            /* number of coefficients */
    int l;            /* interpolation/decimation index */
} fir_state_fx32;
```

Description

The `fir` functions implement a finite impulse response (FIR) filter. The functions 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 are specified by the argument `length`.

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

It is defined as:

```
#define fir_init(state, coeffs, delay, ncoeffs. index) \
                                                \
do {                                            \
    (state).h = (coeffs);    \
```

```
    (state).d = (delay);    \
    (state).p = (delay);    \
    (state).k = (ncoeffs);  \
    (state).l = (index);    \
} while (0)
```

The characteristics of the filter (passband, stopband, and so on) are dependent upon the number of filter coefficients and their values. A pointer to the coefficients should be stored in `filter_state->h`, and `filter_state->k` should be set to the number of coefficients. The functions assume that the coefficients are stored in the normal order, thus `filter_state->h[0]` contains the first filter coefficient and `filter_state->h[k-1]` contains the last coefficient.

> **ATTENTION:** The `fir_fr16` and `fir_fx16` functions will exploit the Blackfin architecture by computing the filtered response of two input samples at one time. As a consequence of this optimization, the input and output vectors and the array of filter coefficients must be aligned on a 32-bit address boundary. Under most circumstances, the compiler will allocate arrays on a 32-bit word-aligned address boundary. However, arrays within structures are not aligned beyond the required alignment for their type. So if any of the input, output, or coefficients arrays are allocated as part of a structure, then they should be explicitly aligned to a word address by preceding their declaration with a `#pragma align 4` directive. For more information, see #pragma align *num* in the Compiler chapter.

Each filter should have its own delay line which is a vector of type `fract16` (for `fir_fr16`), `_Fract` (for `fir_fx16`), `fract32` (for `fir_fr32`) or `long _Fract` (for `fir_fx32`) whose length is equal to the number of coefficients. The vector should be initially cleared to zero and should not otherwise be modified by the user program. The structure member `filter_state->d` should be set to the start of the delay line, and the function uses `filter_state->p` to keep track of its current position within the vector.

The structure member `filter_state->l` is not used by `fir_fr16`, `fir_fx16`, `fir_fr32`, or `fir_fx32`. This field is normally set to an interpolation/decimation index before calling either the `fir_interp` or `fir_decima` functions.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The `fir` functions check that the number of input samples and the number of coefficients are greater than zero-if not, the functions just return.

Algorithm

The following equation is the basis of the algorithm.

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

where:

x = `input`

y = `output`

h = array of coefficients

k = number of coefficients

i = {0, 1, ..., length-1}

Domain

[-1.0 , +1.0)

Example

```
#include <filter.h>

#define NUM_SAMPLES    256
#define NUM_COEFFS      89

fract32 input[NUM_SAMPLES];
fract32 output[NUM_SAMPLES];

#pragma section("L1_data_a")
fract32 coeffs[NUM_COEFFS];

#pragma section("L1_data_b")
fract32 delay[NUM_COEFFS];
fir_state_fr32 state;
int i;

for (i = 0; i < NUM_COEFFS; i++) /* clear the delay line */
{
   delay[i] = 0;
}

fir_init(state, coeffs, delay, NUM_COEFFS, 0);
fir_fr32(input, output, NUM_SAMPLES, &state);
```

fir_decima

FIR decimation filter

Synopsis

```
#include <filter.h>

void fir_decima_fr16(const fract16       input[],
```

```
                        fract16           output[],
                        int               length,
                        fir_state_fr16    *filter_state);

void fir_decima_fx16(const _Fract        input[],
                        _Fract            output[],
                        int               length,
                        fir_state_fx16    *filter_state);

void fir_decima_fr32(const fract32        input[],
                        fract32           output[],
                        int               length,
                        fir_state_fr32    *filter_state);

void fir_decima_fx32(const long _Fract   input[],
                        long _Fract       output[],
                        int               length,
                        fir_state_fx32    *filter_state);
```

The `fir_decima_fr16` function uses the following structure to maintain the state of the filter.

```
typedef struct
{
   fract16 *h;              /* filter coefficients             */
   fract16 *d;              /* start of delay line             */
   fract16 *p;              /* read/write pointer              */
   int k;                   /* number of coefficients          */
   int l;                   /* interpolation/decimation index  */
} fir_state_fr16;
```

The `fir_decima_fx16` function uses the following structure to maintain the state of the filter.

```
typedef struct {
   _Fract *h;               /* filter coefficients             */
   _Fract *d;               /* start of delay line             */
   _Fract *p;               /* read/write pointer              */
   int k;                   /* number of coefficients          */
   int l;                   /* interpolation/decimation index  */
} fir_state_fx16;
```

The `fir_decima_fr32` function uses the following structure to maintain the state of the filter.

```
typedef struct
{
   fract32 *h;              /* filter coefficients             */
   fract32 *d;              /* start of delay line             */
   fract32 *p;              /* read/write pointer              */
   int k;                   /* number of coefficients          */
   int l;                   /* interpolation/decimation index  */
} fir_state_fr32;
```

The `fir_decima_fx32` function uses the following structure to maintain the state of the filter.

CCES 2.9.0 C/C++ Compiler and Library Manual for Blackfin Processors

```
typedef struct
{    long _Fract *h;       /* filter coefficients          */
     long _Fract *d;       /* start of delay line          */
     long _Fract *p;       /* read/write pointer           */
     int k;                /* number of coefficients       */
     int l;                /* interpolation/decimation index */
} fir_state_fx32;
```

Description

The `fir_decima` functions perform an FIR-based decimation filter. They generate the filtered decimated response of the input data `input` and store the result in the output vector `output`. The number of input samples is specified by the argument `length`, and the size of the output vector should be `length/l` where `l` is the decimation index.

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

It is defined as:

```
#define fir_init(state, coeffs, delay, ncoeffs, index) \

do {                                \
    (state).h = (coeffs);    \
    (state).d = (delay);     \
    (state).p = (delay);     \
    (state).k = (ncoeffs);   \
    (state).l = (index);     \
  } while (0)
```

The characteristics of the filter are dependent upon the number of filter coefficients and their values, and on the decimation index supplied by the calling program. A pointer to the coefficients should be stored in `filter_state->h`, and `filter_state->k` should be set to the number of coefficients. The functions assume that the coefficients are stored in the normal order, thus `filter_state->h[0]` contains the first filter coefficient and `filter_state->h[k-1]` contains the last coefficient. The decimation index is supplied to the function in `filter_state->l`.

Each filter should have its own delay line which is a vector of type `fract16` (for `fir_decima_fr16`), `_Fract` (for `fir_decima_fx16`), `fract32` (for `fir_decima_fr32`), or `long _Fract` (for `fir_decima_fx32`) whose length is equal to the number of coefficients. The vector should be initially cleared to zero and should not otherwise be modified by the user program. The structure member `filter_state->d` should be set to the start of the delay line, and the function uses `filter_state->p` to keep track of its current position within the vector.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The `fir_decima` functions check that the number of input samples, the number of coefficients and the decimation index are greater than zero-if not, the functions just return.

Algorithm

The following equation is the basis of the algorithm.

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

where:

h = array of coefficients

k = number of coefficients

n = length

l = decimation index

i = {0, 1, ..., (n / l) - 1}

x = input

y = output

Domain

[-1.0 , +1.0)

Example

```
#include <filter.h>

#define NUM_INSAMPLES    256
#define NUM_COEFFS        89
#define NUM_DECIMATION    16
#define NUM_OUTSAMPLES   (NUM_INSAMPLES / NUM_DECIMATION)

fract32 input[NUM_INSAMPLES];
fract32 output[NUM_OUTSAMPLES];

#pragma section("L1_data_a")
fract32 coeffs[NUM_COEFFS];

#pragma section("L1_data_b")
fract32 delay[NUM_COEFFS];
fir_state_fr32 state;
```

```
int i;

for (i = 0; i < NUM_COEFFS; i++) /* clear the delay line */
{
  delay[i] = 0;
}
fir_init(state, coeffs, delay, NUM_COEFFS, NUM_DECIMATION);
fir_decima_fr32(input, output, NUM_INSAMPLES, &state);
```

fir_interp

FIR interpolation filter

Synopsis

```
#include <filter.h>

void fir_interp_fr16(const fract16     input[],
                     fract16           output[],
                     int               length,
                     fir_state_fr16  *filter_state);

void fir_interp_fx16(const _Fract      input[],
                     _Fract            output[],
                     int               length,
                     fir_state_fx16  *filter_state);

void fir_interp_fr32(const fract32     input[],
                     fract32           output[],
                     int               length,
                     fir_state_fr32  *filter_state);

void fir_interp_fx32(const long _Fract input[],
                     long _Fract       output[],
                     int               length,
                     fir_state_fx32  *filter_state);
```

The `fir_interp_fr16` function uses the following structure to maintain the state of the filter.

```
typedef struct
{    fract16 *h;        /* filter coefficients                 */
     fract16 *d;        /* start of delay line                 */
     fract16 *p;        /* read/write pointer                  */
     int k;             /* number of coefficients per polyphase */
     int l;             /* interpolation/decimation index      */
} fir_state_fr16;
```

The `fir_interp_fx16` function uses the following structure to maintain the state of the filter.

```
typedef struct
{    _Fract *h;         /* filter coefficients                 */
```

```
    _Fract *d;          /* start of delay line                 */
    _Fract *p;          /* read/write pointer                  */
     int k;             /* number of coefficients per polyphase */
     int l;             /* interpolation/decimation index      */
} fir_state_fx16;
```

The `fir_interp_fr32` function uses the following structure to maintain the state of the filter.

```
typedef struct
{    fract32 *h;         /* filter coefficients                 */
     fract32 *d;         /* start of delay line                 */
     fract32 *p;         /* read/write pointer                  */
     int k;              /* number of coefficients per polyphase */
     int l;              /* interpolation/decimation index      */
} fir_state_fr32;
```

The `fir_interp_fx32` function uses the following structure to maintain the state of the filter.

```
typedef struct
{    long _Fract *h;    /* filter coefficients                 */
     long _Fract *d;    /* start of delay line                 */
     long _Fract *p;    /* read/write pointer                  */
     int k;             /* number of coefficients per polyphase */
     int l;             /* interpolation/decimation index      */
} fir_state_fx32;
```

Description

The `fir_interp` functions perform an FIR-based interpolation filter. They generate the interpolated filtered response of the input data `input` and store the result in the output vector `output`. The number of input samples is specified by the argument `length`, and the size of the output vector should be `length*l` where `l` is the interpolation index.

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

The `fir_interp` functions assume that the coefficients are stored in the following order:

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

where:

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

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

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

```
#define fir_init(state, coeffs, delay, ncoeffs, index) \

do {                                      \
            (state).h = (coeffs);    \
            (state).d = (delay);     \
            (state).p = (delay);     \
            (state).k = (ncoeffs);  \
            (state).l = (index);     \
    } while (0)
```

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

Each filter should have its own delay line which is a vector of type `fract16` (for `fir_interp_fr16`), `_Fract` (for `fir_interp_fx16`), `fract32` (for `fir_interp_fr32`), or `long _Fract` (for `fir_interp_fx32`) whose length is equal to the number of coefficients in each polyphase. The vector should be cleared to zero before calling the function for the first time and should not otherwise be modified by the user program. The structure member `filter_state->d` should be set to the start of the delay line, and the function uses `filter_state->p` to keep track of its current position within the vector.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The `fir_interp` functions check that the number of input samples, the number of coefficients and the interpolation index are greater than zero-if not, the functions just return.

Algorithm

The following equation is the basis of the algorithm.

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

where:

h = array of coefficients

k = number of coefficients

n = `length`

l = interpolation index

i = {0, 1, ..., n - 1}

m = {0, 1, ..., l - 1}

x = `input`

y = `output`

Domain

[-1.0 , +1.0)

Example

The following example demonstrates how the library function may be called.

```
#include <filter.h>
#include <fract2float_conv.h>

#define N_INSAMPLES       257
#define N_COEFFS          128
#define N_INTERPOLATION    16
#define N_POLY              N_INTERPOLATION
#define N_COEFFS_PER_POLY (N_COEFFS / N_POLY)
#define N_OUTSAMPLES       (N_INSAMPLES * N_INTERPOLATION)

fract16 signal[N_INSAMPLES];
fract16 output[N_OUTSAMPLES];

/* Filter coefficients from a filter design tool */
float filter_coeffs[N_POLY][N_COEFFS_PER_POLY];

/* Coefficients and delay line for the filter function
   (use separate memory banks for best performance)
*/
#pragma section("L1_data_a") fract16 coeffs[N_COEFFS];
#pragma section("L1_data_b")
fract16 delay[N_COEFFS_PER_POLY];

fir_state_fr16 state;
fract16 x;
int i, np, nc;

/* Transform the coefficients from the filter design tool
   into coefficients for the fir_interp function
   (all filter coefficients are assumed to be < 1.0)
*/
for (np = 0; np < N_POLY; np++) {
   for (nc = 0; nc < N_COEFFS_PER_POLY; nc++) {
      x = float_to_fr16 (filter_coeffs[np][nc]);
         coeffs[(np * N_COEFFS_PER_POLY) + nc] = x;
     }
```

```
}
   /* Configure filter descriptor */
   fir_init (state,coeffs,delay,N_COEFFS_PER_POLY,N_POLY);

/* Zero delay line to start or reset the filter */
for (i = 0; i < N_COEFFS_PER_POLY; i++)
    delay[i] = 0;

/* Perform a FIR-based interpolation filter */
fir_interp_fr16 (signal,output,N_INSAMPLES,&state);
```

gen_bartlett

Generate Bartlett window

Synopsis

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

void gen_bartlett_fx16(_Fract     bartlett_window[],
                       int        window_stride,
                       int        window_size);

void gen_bartlett_fr32(fract32    bartlett_window[],
                       int        window_stride,
                       int        window_size);

void gen_bartlett_fx32(long _Fract bartlett_window[],
                       int        window_stride,
                       int        window_size);
```

Description

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

The Bartlett window is similar to the triangle window (see gen_triangle functions) but has the following different properties:

- The Bartlett window always returns a window with two zeros on either end of the sequence, so that for odd n, the center section of an N+2 Bartlett window equals an N triangle window.

- For even n, the Bartlett window is still the convolution of two rectangular sequences. There is no standard definition for the triangle window for even n; the slopes of the triangle window are slightly steeper than those of the Bartlett window.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equation is the basis of the algorithm.

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

where:

w = `bartlett_window`

N = `window_size`

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

Domain

`window_stride` > 0

N > 0

Example

```
#include <window.h>

#define N  100
#define n  2

fract32 b[n*N];
gen_bartlett_fr32(b, n, N);
```

gen_blackman

Generate Blackman window

Synopsis

```
#include <window.h>
```

```
void gen_blackman_fr16(fract16  blackman_window[],
                       int      window_stride,
                       int      window_size);

void gen_blackman_fr32(fract32  blackman_window[],
                       int      window_stride,
                       int      window_size);

void gen_blackman_fx16(_Fract   blackman_window[],
                       int      window_stride,
                       int      window_size);

void gen_blackman_fx32(long _Fract blackman_window[],
                       int         window_stride,
                       int         window_size);
```

Description

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

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equation is the basis of the algorithm.

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

where:

N = `window_size`

w = `blackman_window`

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

Domain

`window_stride` > 0

N > 0

gen_gaussian

Generate Gaussian window

Synopsis

```
#include <window.h>

void gen_gaussian_fr16(fract16     gaussian_window[],
                       float       alpha,
                       int         window_stride,
                       int         window_size);

void gen_gaussian_fr32(fract32     gaussian_window[],
                       long double alpha,
                       int         window_stride,
                       int         window_size);

void gen_gaussian_fx16(_Fract      gaussian_window[],
                       float       alpha,
                       int         window_stride,
                       int         window_size);

void gen_gaussian_fx32(long _Fract gaussian_window[],
                       long double alpha,
                       int         window_stride,
                       int         window_size);
```

Description

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

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equation is the basis of the algorithm.

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

where:

w = `gaussian_window`

N = `window_size`

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

α is an input parameter

Domain

`window_stride` > 0

`window_size` > 0

α > 0

gen_hamming

Generate Hamming window

Synopsis

```
#include <window.h>

void gen_hamming_fr16(fract16   hamming_window[],
                      int       window_stride,
                      int       window_size);

void gen_hamming_fr32(fract32   hamming_window[],
                      int       window_stride,
                      int       window_size);


void gen_hamming_fx16(_Fract    hamming_window[],
                      int       window_stride,
                      int       window_size);

void gen_hamming_fx32(long _Fract hamming_window[],
                      int         window_stride,
                      int         window_size);
```

Description

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equation is the basis of the algorithm.

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

where:

w = `hamming_window`

N = `window_size`

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

Domain

`window_stride` > 0

N > 0

gen_hanning

Generate Hanning window

Synopsis

```
#include <window.h>

void gen_hanning_fr16(fract16  hanning_window[],
                      int      window_stride,
                      int      window_size);

void gen_hanning_fr32(fract32  hanning_window[],
                      int      window_stride,
                      int      window_size);

void gen_hanning_fx16(_Fract   hanning_window[],
                      int      window_stride,
```

```
                           int         window_size);

void gen_hanning_fx32(long _Fract hanning_window[],
                           int         window_stride,
                           int         window_size);
```

Description

The `gen_hanning` functions generate a vector containing the Hanning window. The length of the window required is specified by the parameter `window_size`, and the parameter `window_stride` is used to space the window values within the output vector `hanning_window`. The length of the output vector should therefore be `window_size*window_stride`. This window is also known as the cosine window.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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

`window_stride` > 0

N > 0

gen_harris

Generate Harris window

Synopsis

```
#include <window.h>

void gen_harris_fr16(fract16  harris_window[],
                          int      window_stride,
                          int      window_size);
```

```
void gen_harris_fr32(fract3   harris_window[],
                     int      window_stride,
                     int      window_size);

void gen_harris_fx16(_Fract   harris_window[],
                     int      window_stride,
                     int      window_size);

void gen_harris_fx32(long _Fract  harris_window[],
                     int          window_stride,
                     int          window_size);
```

Description

The `gen_harris` functions generate a vector containing the Harris window. The length of the window required is specified by the parameter `window_size`, and the parameter `window_stride` is used to space the window values within the output vector `harris_window`. The length of the output vector should therefore be `window_size*window_stride`. This window is also known as the Blackman-Harris window.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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

`window_stride` > 0

N > 0

gen_kaiser

Generate Kaiser window

Synopsis

```
#include <window.h>

void gen_kaiser_fr16(fract16     kaiser_window[],
                     float       beta,
                     int         window_stride,
                     int         window_size);

void gen_kaiser_fr32(fract32     kaiser_window[],
                     long double beta,
                     int         window_stride,
                     int         window_size);

void gen_kaiser_fx16(_Fract      kaiser_window[],
                     float       beta,
                     int         window_stride,
                     int         window_size);

void gen_kaiser_fx32(long _Fract kaiser_window[],
                     long double beta,
                     int         window_stride,
                     int         window_size);
```

Description

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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

$I_0(\beta)$ = Zeroth-order in modified Bessel function of the first kind

Domain

a > 0

N > 0

β > 0.0

gen_rectangular

Generate rectangular window

Synopsis

```
#include <window.h>

void gen_rectangular_fr16(fract16   rectangular_window[],
                          int       window_stride,
                          int       window_size);

void gen_rectangular_fr32(fract32   rectangular_window[],
                          int       window_stride,
                          int       window_size);

void gen_rectangular_fx16(_Fract    rectangular_window[],
                          int       window_stride,
                          int       window_size);

void gen_rectangular_fx32(long _Fract rectangular_window[],
                          int         window_stride,
                          int         window_size);
```

Description

The `gen_rectangle` functions generate a vector containing the rectangular window. The length of the window required is specified by the parameter `window_size`, and the parameter `window_stride` is used to space the window values within the output vector `rectangular_window`. The length of the output vector should therefore be `window_size*window_stride`.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

rectangular_window[n] = 1

where:

N = window_size

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

Domain

window_stride > 0

N > 0

gen_triangle

Generate triangle window

Synopsis

```
#include <window.h>

void gen_triangle_fr16(fract16  triangle_window[],
                       int      window_stride,
                       int      window_size);

void gen_triangle_fr32(fract32  triangle_window[],
                       int      window_stride,
                       int      window_size);

void gen_triangle_fx16(_Fract   triangle_window[],
                       int      window_stride,
                       int      window_size);

void gen_triangle_fx32(long _Fract triangle_window[],
                       int         window_stride,
                       int         window_size);
```

Description

The gen_triangle functions generate a vector containing the triangle window. The length of the window required is specified by the parameter window_size, and the parameter window_stride is used to space the window values within the output vector triangle_window.

Refer to the Bartlett window (gen_bartlett) regarding the relationship between it and the triangle window.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

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

window_stride > 0

N > 0

gen_vonhann

Generate von Hann window

Synopsis

```
#include <window.h>

void gen_vonhann_fr16(fract16  vonhann_window[],
                      int      window_stride,
                      int      window_size);

void gen_vonhann_fr32(fract32  vonhann_window[],
                      int      window_stride,
                      int      window_size);
```

```
void gen_vonhann_fx16(_Fract    vonhann_window[],
                      int       window_stride,
                      int       window_size);

void gen_vonhann_fx32(long _Fract  vonhann_window[],
                      int          window_stride,
                      int          window_size);
```

Description

The gen_vonhann functions are identical to the Hanning window functions (gen_hanning).

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Domain

window_stride > 0

window_size > 0

histogram

Histogram

Synopsis

```
#include <stats.h>

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

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

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

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

void histogram_fx16 (const _Fract   samples[],
                     int            histogram[],
                     _Fract         max_sample,
                     _Fract         min_sample,
                     int            sample_length,
                     int            bin_count);

void histogram_fr32 (const fract32  samples[],
                     int            histogram[],
                     fract32        max_sample,
                     fract32        min_sample,
                     int            sample_length,
                     int            bin_count);

void histogram_fx32 (const long _Fract samples[],
                     int              histogram[],
                     long _Fract      max_sample,
                     long _Fract      min_sample,
                     int              sample_length,
                     int              bin_count);
```

Description

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

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

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

NOTE: To preserve maximum performance while performing out-of-bounds checking, the `histogram_fr16` and histogram_fx16 functions allocate a temporary work area on the stack. The work area is allocated with `(bin_count + 2)` elements and the stack may therefore overflow if the number of bins is sufficiently large. The size of the stack may be adjusted by making appropriate changes to the `.ldf` file.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

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

Domain

[-3.4e38 , +3.4e38]	for `histogramf( )`
[-1.7e308 , +1.7e308]	for `histogramd( )`
[-1.0 , +1.0)	for `histogram_fr16( )`, `histogram_fx16( )`, `histogram_fr32( )`, `histogram_fx32( )`

ifft

Inverse radix-2 Fast Fourier Transform

Synopsis

```
#include <filter.h>

void ifft_fr16(const complex_fract16  input[],
                     complex_fract16  output[],
               const complex_fract16  twiddle_table[],
                     int              twiddle_stride,
                     int              fft_size,
                     int              *block_exponent,
                     int              scale_method);

void ifft_fr32(const complex_fract32  input[],
                     complex_fract32  output[],
               const complex_fract32  twiddle_table[],
                     int              twiddle_size,
                     int              fft_size,
                     int              *block_exponent,
                     int              scale_method);
```

Description

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

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `fft_size/2` twiddle factors. The table is composed of `+cosine` and `-sine` coefficients and may be initialized by using the function `twidfftrad2_fr16` for `ifft_fr16` and `twidfftrad2_fr32` for `ifft_fr32`. For optimal performance, the twiddle table should be allocated in a different memory section than the output array.

The argument `twiddle_stride` should be set to 1 if the twiddle table was originally created for an FFT of size `fft_size`. If the twiddle table was created for a larger FFT of size `N*fft_size` (where `N` is a power of 2), then `twiddle_stride` should be set to `N`. This argument therefore provides a way of using a single twiddle table to calculate FFTs of different sizes.

The argument `scale_method` controls how the function will apply scaling while computing a Fourier Transform. The available options are static scaling (dividing the input at any stage by 2), dynamic scaling (dividing the input at any stage by 2 if the largest absolute input value is greater or equal to 0.25), or no scaling. Note that the number of stages required to compute an FFT is dependent on the size of the FFT and is given by the formula `log2(fft_size)`.

If static scaling is selected, the function always scales intermediate results, thus preventing overflow. The loss of precision increases in line with `fft_size` and is more pronounced for input signals with a small magnitude (since the output is scaled by `1/fft_size`). To select static scaling, set the argument `scale_method` to a value of 1. The block exponent returned will be `log2(fft_size)`.

If dynamic scaling is selected, the function inspects intermediate results and only apply scaling where required to prevent overflow. The loss of precision increases in line with the size of the FFT and is more pronounced for input signals with a large magnitude (since these factors increase the need for scaling). The requirement to inspect intermediate results will have an impact on performance. To select dynamic scaling, set the argument `scale_method` to a value of 2. The block exponent returned will be between 0 and `log2(fft_size)` depending upon the number of times that the function scales each set of intermediate results.

If no scaling is selected, the function does not scale intermediate results. There is no loss of precision unless overflow occurs and in this case the function will generate saturated results. The likelihood of saturation increases in line with the `fft_size` and is more pronounced for input signals with a large magnitude. To select no scaling, set the argument `scale_method` to 3. The block exponent returned will be 0.

NOTE: Any values for the argument `scale_method` other than 2 or 3 will result in the function performing static scaling.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The `ifft` functions abort if the FFT size is less than 8 or if the twiddle stride is less than 1.

Algorithm

The following equation is the basis of the algorithm.

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

Domain

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

Example

```
/* Compute  IFFT( CFFT( X ) ) = X */
#include <filter.h>
#define N_FFT  64

complex_fract16   in[N_FFT];
complex_fract16   out_cfft[N_FFT];
complex_fract16   out_ifft[N_FFT];
complex_fract16   twiddle[N_FFT/2];
int               blk_exp;

void ifft_fr16_example(void)
{
   int i;
   /* Generate DC signal */
   for( i = 0; i < N_FFT; i++ )
   {
      in[i].re = 0x100;
        in[i].im = 0x0;    }
/* Populate twiddle table */
twidfftrad2_fr16(twiddle, N_FFT);

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

/* Reverse static scaling applied by cfft_fr16() function
   Apply the shift operation before the call to the ifft_fr16() function
   only if all the values in out_cfft = 0x100.
   Otherwise, perform the shift operation after the ifft_fr16() function
   has been computed.
*/
for( i = 0; i < N_FFT; i++ )
{
   out_cfft[i].re = out_cfft[i].re << 6; /* log2(N_FFT) = 6 */
   out_cfft[i].im = out_cfft[i].im << 6;
}
/* Compute Inverse Fast Fourier Transform
   The output signal from the ifft function will be the same
   as the DC signal of magnitude 0x100 which was passed into
```

```
   the cfft function.
*/
ifft_fr16(out_cfft, out_ifft, twiddle, 1, N_FFT, &blk_exp, 0);
}
```

ifftf

Fast Inverse N-point Fast Fourier Transform

Synopsis

```
#include <filter.h>
void ifftf_fr16(const complex_fract16 input[],
                      complex_fract16 output[],
               const complex_fract16 twiddle_table[],
                      int             twiddle_stride,
                      int             fft_size);
void ifftf_fr32(const complex_fract32 input[],
                      complex_fract32 output[],
               const complex_fract32 twiddle_table[],
                      int             twiddle_stride,
                      int             fft_size);
```

Description

The `ifftf` functions transform 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. The functions use a mixed-radix algorithm.

The size of the input array `input` and the output array `output` is `fft_size`, where `fft_size` represents the number of points in the FFT. The number of points in the FFT must be a power of 2 and must be at least 8.

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `3*fft_size/4` complex twiddle factors. The table should be initialized with complex twiddle factors in which the real coefficients are positive cosine values and the imaginary coefficients are negative sine values. The function `twidfftf_fr16` may be used to initialize the array for `ifftf_fr16`, while the `twidfftf_fr32` function may be used to initialize the array for `ifftf_fr32`.

If the twiddle table has been generated for an `fft_size` FFT, then the `twiddle_stride` argument should be set 1. On the other hand, if the twiddle table has been generated for an FFT of size x, where `x > fft_size`, then the `twiddle_stride` argument should be set to `x/fft_size`. The `twiddle_stride` argument therefore allows the same twiddle table to be used for different sizes of FFT. (The `twiddle_stride` argument cannot be either zero or negative).

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

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equation is the basis of the algorithm.

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

The functions use a mixed-radix algorithm (radix-4 and radix-2).

Example

```
#include <filter.h>

#define FFT_SIZE1 32
#define FFT_SIZE2 256
#define TWID_SIZE ((3 * FFT_SIZE2) / 4)

complex_fract32  in1[FFT_SIZE1], in2[FFT_SIZE2];
complex_fract32  out1[FFT_SIZE1], out2[FFT_SIZE2];
complex_fract32  twiddle[TWID_SIZE];

twidfftf_fr32(twiddle,FFT_SIZE2);

ifftf_fr32(in1, out1, twiddle, FFT_SIZE2/FFT_SIZE1, FFT_SIZE1);

ifftf_fr32(in2, out2, twiddle, 1, FFT_SIZE2);
```

ifft2d

N x N point 2-D inverse input FFT

Synopsis

```
#include <filter.h>

void ifft2d_fr16(const complex_fract16 *input,
                       complex_fract16 *temp,
                       complex_fract16 *output,
                 const complex_fract16  twiddle_table[],
                       int              twiddle_stride,
                       int              fft_size,
```

```
                        int              block_exponent,
                        int              scale_method);

void ifft2d_fr32(const complex_fract32 *input,
                        complex_fract32 *temp,
                        complex_fract32 *output,
              const complex_fract32  twiddle_table[],
                        int              twiddle_stride,
                        int              fft_size);
```

Description

The `ifft2d` functions compute a two-dimensional Inverse Fast Fourier Transform (FFT) of the complex input matrix `input[fft_size][fft_size]` and store the result to the complex output matrix `output[fft_size][fft_size]`.

The size of the input array `input`, the output array `output`, and the temporary working buffer `temp` is `fft_size*fft_size`, where `fft_size` represents the number of points in the FFT. The argument `fft_size` must be a power of 2 and must be at least 4 for `ifft2d_fr16` and at least 8 for `ifft2d_fr32`.

Memory bank collisions, which have an adverse effect on run-time performance, may be avoided by allocating the temporary array and the twiddle table in separate memory banks if using `ifft2d_fr16`, or by allocating the twiddle table in a different memory bank than the output array and the temporary array if using `ifft2d_fr32`.

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `fft_size` twiddle factors for `ifft2d_fr16` and at least `3*fft_size/4` twiddle factors for `ifft2d_fr32`. The table should be initialized with complex twiddle factors in which the real coefficients are positive cosine values and the imaginary coefficients are negative sine values. The functions `twidfft2d_fr16` and `twidfft2d_fr32` may be used to initialize the arrays for `ifft2d_fr16` and `ifft2d_fr32` respectively.

If the twiddle table has been generated for an `fft_size` FFT, the `twiddle_stride` argument should be set 1. On the other hand, if the twiddle table has been generated for an FFT of size x, where x>`fft_size`, then the `twiddle_stride` argument should be set to x/`fft_size`. The `twiddle_stride` argument therefore allows the same twiddle table to be used for different sizes of FFT. (The `twiddle_stride` argument cannot be either zero or negative).

To avoid overflow, the functions scale the output by `fft_size * fft_size`.

The `ifft2d_fr16` arguments `block_exponent` and `scale_method` have been added for future expansion. These arguments are ignored by the function.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The `ifft2d` functions abort if the twiddle stride is less than 1, or if `fft_size` is less than 4 for `ifft2d_fr16`, or if `fft_size` is less than 8 for `ifft2d_f32`.

Algorithm

The following equation is the basis of the algorithm.

$$c(i, j) = \frac{1}{n^2} \cdot \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} a(k, l) \cdot e^{-2\pi j(i \cdot k + j \cdot l)/n}$$

where:

i = {0, 1, ..., n-1}

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

Domain

Input sequence length `fft_size` must be a power of 2 and at least 4 for `ifft2d_fr16` and at least 8 for `ifft2d_fr32`.

Example

```
#include <filter.h>

#define FFT_SIZE1          128
#define FFT_SIZE2           32
#define TWIDDLE_STRIDE1  (FFT_SIZE1 / FFT_SIZE1)
#define TWIDDLE_STRIDE2  (FFT_SIZE1 / FFT_SIZE2)

complex_fract32  in1[FFT_SIZE1][FFT_SIZE1];
complex_fract32  in2[FFT_SIZE2][FFT_SIZE2];
complex_fract32  out2[FFT_SIZE2][FFT_SIZE2];
complex_fract32  tmp[FFT_SIZE1][FFT_SIZE1];
complex_fract32  twiddle[(3*FFT_SIZE1)/4];

twidfft2d_fr32 (twiddle, FFT_SIZE1);

/* In-place computation */
ifft2d_fr32(in1, tmp, in1, twiddle, TWIDDLE_STRIDE1, FFT_SIZE1);
ifft2d_fr32(in2, tmp, out2, twiddle, TWIDDLE_STRIDE2, FFT_SIZE2);
```

iir

Infinite impulse response filter

Synopsis

```
#include <filter.h>
#include <filter.h>

void iir_fr16(const fract16    input[],
              fract16          output[],
              int              length,
              iir_state_fr16   *filter_state);

void iir_fx16(const _Fract     input[],
              _Fract           output[],
               int             length,
              iir_state_fx16   *filter_state);

void iir_fr32(const fract32    input[],
              fract32          output[],
              int              length,
              iir_state_fr32   *filter_state);

void iir_fx32(const long _Fract input[],
              long _Fract output[],
              int         length,
              iir_state_fx32   *filter_state);
```

The `iir_fr16` function uses the following structure to maintain the state of the filter.

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

The `iir_fr16` function uses the following structure to maintain the state of the filter.

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

The `iir_fx16` function uses the following structure to maintain the state of the filter.

```
typedef struct
{
   _Fract *c;           /* coefficients             */
   _Fract *d;           /* start of delay line      */
    int k;              /* number of biquad stages  */
} iir_state_fx16;
```

The `iir_fr32` function uses the following structure to maintain the state of the filter.

```
typedef struct
{
    fract32 *c;          /* coefficients          */
    fract32 *d;          /* start of delay line    */
    int k;               /* number of biquad stages */
} iir_state_fr32;
```

The `iir_fx32` function uses the following structure to maintain the state of the filter.

```
typedef struct
{
    long _Fract *c;      /* coefficients          */
    long _Fract *d;      /* start of delay line    */
    int k;               /* number of biquad stages */
} iir_state_fx32;
```

Description

The `iir` functions implement a biquad direct form II infinite impulse response (IIR) filter. They 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 are specified by the argument `length`.

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

```
#define iir_init(state, coeffs, delay, stages)  \
do {                                             \
    (state).c = (coeffs);                        \
    (state).d = (delay);                         \
    (state).k = (stages);                        \
} while (0)
```

The characteristics of the filter are dependent upon filter coefficients and the number of stages. Each stage has five coefficients which must be stored in the order `A2`, `A1`, `B2`, `B1`, and `B0`. The value of `A0` is implied to be `1.0` and `A1` and `A2` should be scaled accordingly. This requires that the value of the `A0` coefficient be greater than both `A1` and `A2` for all the stages. The functions `iirdf1_fr16`, `iirdf1_fx16`, `iirdf1_fr32`, and `iirdf1_fx32` (see iirdf1) implement a direct form I filter, and do not impose this requirement; however, they do assume that the `A0` coefficients are 1.0.A pointer to the coefficients should be stored in `filter_state->c`, and `filter_state->k` should be set to the number of stages.

Each filter should have its own delay line which is a vector of type `fract16` (for `iir_fr16`), `_Fract` (for `iir_fx16`), `fract32` (for `iir_fr32`), or `long _Fract` (for `iir_fx32`), whose length is equal to twice the number of stages. The vector should be initially cleared to zero and should not otherwise be modified by the user program. The structure member `filter_state->d` should be set to the start of the delay line.

> **ATTENTION:** The `iir_fr16` and `iir_fx16` functions will exploit the Blackfin architecture by computing the filtered response of two input samples at one time. As a consequence of this optimization, the input

and output vectors and delay line must be aligned on a 32-bit address boundary. Under most circumstances, the compiler will allocate arrays on a 32-bit word-aligned address boundary. However, arrays within structures are not aligned beyond the required alignment for their type. So if any of the input or output arrays, or the delay line, are allocated as part of a structure, then they should be explicitly aligned to a word address by preceding their declaration with a `#pragma align 4` directive. For more information, see #pragma align *num* in the *Compiler* chapter.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equation is 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

$$D_m = X_m - A_2 \cdot D_{m-2} - A_1 \cdot D_{m-1}$$

$$Y_m = B_2 \cdot D_{m-2} + B_1 \cdot D_{m-1} + B_0 \cdot D_m$$

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

Domain

[-1.0 , +1.0)

Example

```
#include <filter.h>
#include <fract2float_conv.h>

#define NUM_STAGES    2
#define NUM_SAMPLES   64

/* Filter coefficients generated by a filter design
   tool that uses a direct form II    */

const struct {
    float a0;
    float a1;
    float a2;
```

```
} A_coeffs[NUM_STAGES] = {
      1.000000F, 0.453120F, 0.466326F,
      1.000000F, 0.328976F, 0.064588F,
};

const struct {
   float b0;
   float b1;
   float b2;
} B_coeffs[NUM_STAGES] = {
       1.000000F, -2.000000F, 1.000000F,
       1.000000F, -2.000000F, 1.000000F,
};

const int Bscale = 2; /* to scale B-coeffs into the fract */
                      /* range (must be a power of 2)     */

/* Coefficients and delay line for the iir function
   (use separate memory banks for best performance)*/

#pragma section("L1_data_a")fract16 coeffs[NUM_STAGES * 5];
#pragma section("L1_data_b")fract16 delay[NUM_STAGES * 2];

iir_state_fr16 filter_state;

/* Input and output arrays */
fract16 signal[NUM_SAMPLES];
fract16 output[NUM_SAMPLES];
int k;

/* Transform the A-coefficients and B-coefficients from a
   filter design tool into the form required by iir_fr16
   -> A0 coefficients are assumed to be 1.0, and are not
   passed to the iir function
   -> A1 and A2 coefficients must be scaled against the A0
   coefficient (use the iirdf1_fr16 function instead if
   the A1 and A2 coefficients are larger than A0)
   -> scale the B coefficients to fit into the fractional
   range [-1..1]; the scale factor must be a power of 2
*/
for (k = 0; k < NUM_STAGES; k++) {
    coeffs[(5*k)+0] = float_to_fr16 (A_coeffs[k].a2);
    coeffs[(5*k)+1] = float_to_fr16 (A_coeffs[k].a1);
    coeffs[(5*k)+2] = float_to_fr16 (B_coeffs[k].b2/Bscale);
    coeffs[(5*k)+3] = float_to_fr16 (B_coeffs[k].b1/Bscale);
    coeffs[(5*k)+4] = float_to_fr16 (B_coeffs[k].b0/Bscale);}

/* Configure filter state */i
ir_init (filter_state,coeffs,delay,NUM_STAGES);
/* Zero delay line to start or reset the filter */
```

```
for (k = 0; k < (NUM_STAGES * 2); k++)
   delay[k] =0;

/* Compute filter response */
iir_fr16 (signal,output,NUM_SAMPLES,&filter_state);
/* Undo scaling B coefficients */
for (k = 0; k < NUM_SAMPLES; k++)
   output[k] = output[k] * powf (Bscale, NUM_STAGES);
```

iirdf1

Direct form I impulse response filter

Synopsis

```
#include <filter.h>

void iirdf1_fr16(const fract16           input[],
                       fract16           output[],
                       int               length,
                       iirdf1_state_fr16 *filter_state);

void iirdf1_fx16(const _Fract            input[],
                       _Fract            output[],
                       int               length,
                       iirdf1_state_fx16 *filter_state);

void iirdf1_fr32(const fract32           input[],
                       fract32           output[],
                       int               length,
                       iirdf1_state_fr32 *filter_state);

void iirdf1_fx32(const long _Fract       input[],
                       long _Fract       output[],
                       int               length,
                       iirdf1_state_fx32 *filter_state);
```

The `iirdf1_fr16` function uses the following structure to maintain the state of the filter.

```
typedef struct
{
    fract16 *c;     /* coefficients                      */
    fract16 *d;     /* start of delay line               */
    fract16 *p;     /* read/write pointer                */
    int k;          /* 2*number of stages + 1            */
} iirdf1_state_fr16;
```

The `iirdf1_fx16` function uses the following structure to maintain the state of the filter.

```
typedef struct
{
```

```
    _Fract *c;        /* coefficients                    */
    _Fract *d;        /* start of delay line             */
    _Fract *p;        /* read/write pointer              */
    int k;            /* 2*number of stages + 1          */
} iirdf1_state_fx16;
```

The `iirdf1_fr32` function uses the following structure to maintain the state of the filter.

```
typedef struct
{
    fract32 *c;     /* coefficients                      */
    fract32 *d;     /* start of delay line               */
    fract32 *p;     /* read/write pointer                */
    int k;          /* 2*number of stages + 1            */
} iirdf1_state_fr32;
```

The `iirdf1_fx32` function uses the following structure to maintain the state of the filter.

```
typedef struct
{
    long _Fract *c;    /* coefficients                   */
    long _Fract *d;    /* start of delay line            */
    long _Fract *p;    /* read/write pointer             */
    int k;             /* 2*number of stages + 1         */
} iirdf1_state_fx32;
```

Description

The `iirfd1` functions implement a direct form I infinite impulse response (IIR) filter. The functions 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 `length`.

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

The macro is defined as:

```
#define iirdf1_init(state, coeffs, delay, stages) \
do {                                               \
        (state).c = (coeffs);                      \
        (state).d = (delay);                       \
        (state).p = (delay);                       \
        (state).k = (2*(stages)+1);                \
} while (0)
```

The characteristics of the filter are dependent upon the filter coefficients and the number of stages. The A- and B-coefficients for each stage are stored in a vector that is addressed by the pointer `filter_state->c`. This vector is generated by:

- The `coeff_iirdf1_fr16` function for use with `iirdf1_fr16`

- The `coeff_iirdf1_fx16` function for use with `iirdf1_fx16`

- The `coeff_iirdf1_fr32` function for use with `iirdf1_fr32`

- The `coeff_iirdf1_fx32` function for use with `iirdf1_fx32`

Refer to coeff_iirdf1 for more information. The variable `filter_state->k` should be set to the expression `(2*stages) + 1`.

> **NOTE:** Each of the `iirdf1` and `iir` functions assume that the value of the `A0` coefficients is 1.0, and that all other A-coefficients have been scaled according. For the `iir` functions, this also implies that the value of the `A0` coefficient is greater than both the `A1` and `A2` for all stages. This restriction does not apply to the `iirdf1` functions because the coefficients are specified as floating-point values to the `coeff_iirdf1` functions.

Each filter should have its own delay line which is a vector of type `fract16` (for `iirdf1_fr16`) or `_Fract` (for `iirdf1_fr16`), `fract32` (for `iirdf1_fr32`), or `long _Fract` (for `iirdf1_fx32`) whose length is equal to `(4 * stages) + 2`. The vector should be initially cleared to zero and not otherwise modified by the user program. The structure member `filter_state->d` should be set to the start of the delay line, and the function uses `filter_state->p` to keep track of its current position within the vector. For optimum performance, coefficient and state arrays should be allocated in separate memory blocks.

The `iirdf1` functions adjust the output by the scaling factor that is applied to the A- and B-coefficients by the `coeff_iirdf1` functions.

> **NOTE:** It is possible the filter's gain causes the filtered response to be saturated. To avoid the saturation, the B-coefficients can be scaled *before* calling the `coeff_iirdf1` functions; refer to the following example.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for more information.

Algorithm

The following equation is 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:

$V = B_0 * x(i) + B_1 * x(i-1) + B_2 * x(i-2)$

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

$i = \{0, 1, .., \text{length-1}\}$

x = `input`

y = output

Domain

[-1.0 , +1.0)

Example

```
#include <filter.h>
#include <vector.h>

#define NSAMPLES 50
#define NSTAGES  2

/* Coefficients for the coeff_iirdf1_fr16 function */
const float a_coeffs[(2 * NSTAGES)] = { . . . };
const float b_coeffs[(2 * NSTAGES) + 1] = { . . . };
float *coeffs = (float *)b_coeffs;

/* Coefficients for the iirdf1_fr16 function */
fract16 df1_coeffs[(4 * NSTAGES) + 2];

/* Input, Output, Delay Line, and Filter State */
fract16 input[NSAMPLES], output[NSAMPLES];
fract16 delay[(4 * NSTAGES) + 2];
iirdf1_state_fr16 state;
float gain;
int i;

/* Initialize filter description */
iirdf1_init (state,df1_coeffs,delay,NSTAGES);

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

/* Convert coefficients */
if (gain >= 1.0F)
   { vecsmltf (b_coeffs,(1.0F/gain), b_coeffs,((2*NSTAGES)+1)); }
coeff_iirdf1_fr16 (a_coeffs,b_coeffs,df1_coeffs,NSTAGES);

/* Call the function */
iirdf1_fr16 (input,output,NSAMPLES,&state);
```

max

Maximum

Synopsis

```
#include <math.h>

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

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

fract16 max_fr16 (fract16 parm1, fract16 parm2);
fract32 max_fr32 (fract32 parm1, fract32 parm2);

_Fract max_fx16 (_Fract parm1, _Fract parm2);
long _Fract max_fx32 (long _Fract parm1, long _Fract parm2);
```

Description

The max functions return the larger of their two arguments.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

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

Domain

Full range for type of parameters

mean

Mean

Synopsis

```
#include <stats.h>

float meanf(const float samples[], int sample_length);
double mean(const double samples[], int sample_length);
long double meand(const long double samples[], int sample_length);
fract16 mean_fr16(const fract16 samples[], int sample_length);
_Fract mean_fx16(const _Fract samples[], int  sample_length);
```

```
fract32 mean_fr32(const fract32 samples[], int sample_length);
long _Fract mean_fx32(const long _Fract  samples[], int sample_length);
```

Description

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equation is the basis of the algorithm.

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

Error Conditions

The mean_fr16 and mean_fx16 functions can be used to compute the mean of up to 65535 input data with a value of 0x8000 before the sum a_i saturates. The mean_fr32 and mean_fx32 functions can be used to compute the mean of up to 4294967295 input data with a value of 0x80000000 before the sum a_i saturates.

Domain

[-3.4e38 , +3.4e38]	for meanf( )
[-1.7e308 , +1.7e308]	for meand( )
[-1.0 , +1.0)	for mean_fr16( ), mean_fx16( ), mean_fr32( ), mean_fx32( )

min

Minimum

Synopsis

```
#include <math.h>

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

float fminf (float parm1, float parm2);
double fmin (double parm1, double parm2);
```

```
long double fmind (long double parm1, long double parm2);

fract16 min_fr16 (fract16 parm1, fract16 parm2);
fract32 min_fr32 (fract32 parm1, fract32 parm2);

_Fract min_fx16 (_Fract parm1, _Fract parm2);
long _Fract min_fx32 (long _Fract parm1, long _Fract parm2);
```

Description

The `min` functions return the smaller of their two arguments.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

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

Domain

Full range for type of parameters used.

mu_compress

μ-law compression

Synopsis

```
#include <filter.h>

void mu_compress(const short  input[],
                 short        output[],
                 int          length);
```

Description

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

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

Domain

Content of input array: [-8192 , 8191]

mu_expand

μ-law expansion

Synopsis

```
#include <filter.h>

void mu_expand(const short   input[],
               short         output[],
               int           length);
```

Description

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

C(k)= mu_law expansion of A(*k*) for k = 0 to length-1

Domain

Content of input array: [0 , 255]

norm

Normalization

Synopsis

```
#include <complex.h>

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

Description

The normalization functions normalize the complex input a and return the result.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equations are the basis of the algorithm.

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

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

Domain

[-3.4e38 , +3.4e38]	for `normf( )`
[-1.7e308 , +1.7e308]	for `normd( )`

polar

Convert polar to Cartesian notation

Synopsis

```
#include <complex.h>

complex_float polarf(float   magnitude,
                     float   phase);

complex_double polar(double   magnitude,
                     double   phase);

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

complex_fract16 polar_fr16(fract16   magnitude,
                           fract16   phase);

complex_fract32 polar_fr32(fract32   magnitude,
                           fract32   phase);

complex_fract16 polar_fx_fr16(_Fract    magnitude,
```

```
                                  _Fract      phase);

complex_fract32 polar_fx_fr32(long _Fract     magnitude,
                              long _Fract     phase);
```

Description

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

The phase must be scaled by 2π and must be in the range `[0x8000, 0x7fff]` for the `polar_fr16` and `polar_fx_fr16` functions, and in the range `[0x80000000, 0x7fffffff]` for the `polar_fr32` and `polar_fx_fr32` functions. The value of the phase may be either positive or negative. Positive values are interpreted as an anti-clockwise motion around a circle with a radius equal to the magnitude as shown in the *Positive and Negative Phases for Fractional Polar Functions* table. Negative values for the phase argument are interpreted as a clockwise movement.

Table 5-10: Positive and Negative Phases for Fractional Polar Functions

Radians	Phase	
0	0.0	-1
π/2	0.25(0x2000)	-0.75
π	0.50(0x4000)	-0.5
3/2π	0.75(0x6000)	-0.25
<2π	0.999(0x7fff)	

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equations are the basis of the algorithm.

`Re(c) = r*cos(θ)`

`Im(c) = r*sin(θ)`

where:

θ is the phase

t is the magnitude

Domain

phase = [-1.0294e+5, 1.0294e+5] magnitude = [-3.4e38 , +3.4e38]	for `polarf ( )`
phase = [-8.43315e8 , 8.43315e8] magnitude = [-1.7e308 , +1.7e308]	for `polard ( )`
[-1.0 , +1.0)	for `polar_fr16( )`, `polar_fx_fr16( )`, `polar_fr32( )` and `polar_fx_fr32( )`

Example

```
#include <complex.h>
#include <fract2float_conv.h>

#define PI 3.14159265

complex_fract16 point;
float phase_float;

fract16 phase_fr16;
fract16 mag_fr16;

phase_float = PI;
phase_fr16 = float_to_fr16(phase_float  / (2*PI));
mag_fr16 = 0x0200;

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

rfft

Real radix-2 Fast Fourier Transform

Synopsis

```
#include <filter.h>

void rfft_fr16(const fract16          input[],
               complex_fract16        output[],
               const complex_fract16  twiddle_table[],
               int                    twiddle_stride,
               int                    fft_size,
               int                   *block_exponent,
               int                    scale_method);

void rfft_fx_fr16(const _Fract           input[],
                  complex_fract16        output[],
                  const complex_fract16  twiddle_table[],
                  int                    twiddle_stride,
```

```
                            int                 fft_size,
                            int                 *block_exponent,
                            int                  scale_method);


void rfft_fr32(const fract32          input[],
               complex_fract32        output[],
               const complex_fract32  twiddle_table[],
               int                    twiddle_stride,
               int                    fft_size,
               int                    *block_exponent,
               int                    scale_method);


void rfft_fx_fr32(const long _Fract    input[],
               complex_fract32        output[],
               const complex_fract32  twiddle_table[],
               int                    twiddle_stride,
               int                    fft_size,
               int                    *block_exponent,
               int                    scale_method);
```

Description

The `rfft` functions transform the time domain real input signal sequence to the frequency domain by using the radix-2 FFT. The functions take advantage of the fact that the imaginary part of the input equals zero, which in turn eliminates half of the multiplications in the butterfly.

The size of the input array `input` and the output array `output` is `fft_size`, where `fft_size` represents the number of points in the FFT. By allocating these arrays in different memory banks, any potential data bank collisions are avoided, thus improving run-time performance. If the input data can be overwritten, the optimum memory usage can be achieved by also specifying the input array as the output array, provided that the memory size of the input array is at least `2*fft_size`.

The twiddle table is passed in the argument `twiddle_table`, which must contain at least `fft_size/2` twiddle factors. The table is composed of `+cosine` and `-sine` coefficients and may be initialized by using the function `twidfftrad2_fr16` for use with `rfft_fr16` or `rfft_fx_fr16`, and `twidfftrad2_fr32` for use with `rfft_fr32` or `rfft_fx_fr32`. For optimal performance, the twiddle table should be allocated in a different memory section than the output array.

The argument `twiddle_stride` should be set to 1 if the twiddle table was originally created for an FFT of size `fft_size`. If the twiddle table was created for a larger FFT of size `N*fft_size` (where N is a power of 2), then `twiddle_stride` should be set to N. This argument therefore provides a way of using a single twiddle table to calculate FFTs of different sizes.

The argument `scale_method` controls how the functions will apply scaling while computing a Fourier Transform. The available options are static scaling (dividing the input at any stage by 2), dynamic scaling (dividing the input at any stage by 2 if the largest absolute input value is greater or equal than 0.25), or no scaling. Note that the

number of stages required to compute an FFT is dependent on the size of the FFT and is given by the formula `log2(fft_size)`.

If static scaling is selected, the functions will always scale intermediate results, thus preventing overflow. The loss of precision increases in line with `fft_size` and is more pronounced for input signals with a small magnitude (since the output is scaled by `1/fft_size`). To select static scaling, set the argument `scale_method` to a value of 1. The block exponent returned will be `log2(fft_size)`.

If dynamic scaling is selected, the functions inspect intermediate results and only apply scaling where required to prevent overflow. The loss of precision increases in line with the size of the FFT and is more pronounced for input signals with a large magnitude (since these factors increase the need for scaling). The requirement to inspect intermediate results will have an impact on performance. To select dynamic scaling, set the argument `scale_method` to a value of 2. The block exponent returned will be between 0 and `log2(fft_size)`, depending upon the number of times that the functions scales the intermediate set of results.

If no scaling is selected, the functions will never scale intermediate results. There will be no loss of precision unless overflow occurs and in this case the functions will generate saturated results. The likelihood of saturation increases in line with the `fft_size` and is more pronounced for input signals with a large magnitude. To select no scaling, set the argument `scale_method` to 3. The block exponent returned will be 0.

> **NOTE:** Any values for the argument `scale_method` other than 2 or 3 will result in the functions performing static scaling.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The `rfft` functions abort if the FFT size is less than 8 or if the twiddle stride is less than 1.

Algorithm

See cfft for more information.

Domain

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

Example

```
#include <filter.h>

#define FFT_SIZE1    32
#define FFT_SIZE2   256
#define TWID_SIZE  (FFT_SIZE2/2)

fract32          in1[FFT_SIZE1],  in2[FFT_SIZE2];
```

```
complex_fract32  out1[FFT_SIZE1], out2[FFT_SIZE2];
complex_fract32  twiddle[TWID_SIZE];
int block_exponent1, block_exponent2;

twidfftrad2_fr32 (twiddle, FFT_SIZE2);

rfft_fr32 (in1, out1, twiddle, (FFT_SIZE2 / FFT_SIZE1), FFT_SIZE1,
           &block_exponent1, 1 /*static scaling*/ );

rfft_fr32 (in2, out2, twiddle, 1, FFT_SIZE2, &block_exponent2,
           2 /*dynamic scaling*/ );
```

rfftf

Fast N-point real input Fast Fourier Transform

Synopsis

```
#include <filter.h>

void rfftf_fr16(const fract16          input[],
                complex_fract16        output[],
                const complex_fract16 twiddle_table[],
                int                    twiddle_stride,
                int                    fft_size);
void rfftf_fr32(const fract32          input[],
                complex_fract32        output[],
                const complex_fract32 twiddle_table[],
                int                    twiddle_stride,
                int                    fft_size);
void rfftf_fx_fr16(const _Fract          input[],
                   complex_fract16       output[],
                   const complex_fract16 twiddle_table[],
                   int                   twiddle_stride,
                   int                   fft_size);
void rfftf_fx_fr32(const long _Fract     input[],
                   complex_fract32       output[],
                   const complex_fract32 twiddle_table[],
                   int                   twiddle_stride,
                   int                   fft_size);
```

Description

The `rfftf` functions transform the time domain real 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. They decimate in frequency using a mixed-radix algorithm.

The size of the input array `input` and the output array `output` is `fft_size`, where `fft_size` represents the number of points in the FFT. The number of points in the FFT must be a power of 2 and must be at least 16.

As the complex spectrum of a real FFT is symmetrical about the midpoint, the rfftf functions only generate the first `(fft_size/2)+1` points of the FFT, and so the size of the output array `output` must be at least of length `(fft_size/2) + 1`. After returning, the output array contains the following values:

- DC component of the signal in `output[0].re` (`output[0].im = 0`)

- First half of the complex spectrum in `output[1] ...output[(fft_size/2)-1]`

- Nyquist frequency in `output[fft_size/2].re` (with `output[fft_size/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 twiddle table is passed in the argument `twiddle_table`, which must contain at least `3*fft_size/4` complex twiddle factors. The table should be initialized with complex twiddle factors in which the real coefficients are positive cosine values and the imaginary coefficients are negative sine values. The function twidfftf_fr16 may be used to initialize the array for `rfftf_fr16` and `rfftf_fx_fr16`, and the function `twidfftf_fr32` may be used to initialize the array for `rfftf_fr32` and `rfftf_fx_fr32`.

If the twiddle table has been generated for an `fft_size` FFT, then the `twiddle_stride` argument should be set 1. On the other hand, if the twiddle table has been generated for an FFT of size x, where `x > fft_size`, then the `twiddle_stride` argument should be set to `x / fft_size`. The `twiddle_stride` argument therefore allows the same twiddle table to be used for different sizes of FFT. (The `twiddle_stride` argument cannot be either zero or negative).

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

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

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equation is the basis of the algorithm.

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

The implementation uses a mixed-radix algorithm (radix4 and radix-2).

Example

```
#include <filter.h>
#include <complex.h>
```

```
#define FFTSIZE   32
#define TWIDSIZE ((3 * FFTSIZE) / 4)

fract32   sigdata[FFTSIZE];
complex_fract32   r_output[FFTSIZE];
complex_fract32   twiddles[TWIDSIZE];
int i;

/* Initialize the twiddle table */
twidfftf_fr32(twiddles,FFTSIZE);

/* Calculate the FFT of a real signal */
rfftf_fr32(sigdata, r_output, twiddles,1,FFTSIZE);

/* rfftf_fr32 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] = conj_fr32(r_output[i]);
}
```

rfft2d

N x N point 2-D real input FFT

Synopsis

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

void rfft2d_fx_fr16(const _Fract            input[],
                    complex_fract16         temp[],
                    complex_fract16         output[],
                    const complex_fract16   twiddle_table[],
                    int                     twiddle_stride,
                    int                     fft_size,
                    int                     block_exponent,
                    int                     scale_method);

void rfft2d_fr32(const fract32           input[],
                 complex_fract32         temp[],
                 complex_fract32         output[],
```

```
                  const complex_fract32      twiddle_table[],
                  int                        twiddle_stride,
                  int                        fft_size);


void rfft2d_fx_fr32(const long _Fract        input[],
                  complex_fract32            temp[],
                  complex_fract32            output[],
                  const complex_fract32      twiddle_table[],
                  int                        twiddle_stride,
                  int                        fft_size);
```

Description

The `rfft2d` functions compute a two-dimensional Fast Fourier Transform (FFT) of the real input matrix `input[fft_size][fft_size]`, and store the result to the complex output matrix `output[fft_size][fft_size]`.

The size of the input array `input`, the output array `output`, and the temporary working buffer `temp` is `fft_size*fft_size`, where `fft_size` represents the number of rows and number of columns in the FFT. The argument `fft_size` must be a power of 2 and must be at least 4 for `rfft2d_fr16` and `rfft2d_fx_fr16`, and at least 16 for `rfft2d_fr32` and `rfft2d_fx_fr32`.

Memory bank collisions, which have an adverse effect on run-time performance, may be avoided by allocating the temporary array and the `twiddle` table in separate memory banks if using `rfft2d_fr16`, or by allocating the `twiddle` table in a different memory bank than the output array and the temporary array if using `rfft2d_fr32`. If the input data can be overwritten, optimal memory usage can be achieved by also specifying the input matrix as the output buffer, provided that the size of the input array is at least `2 * fft_size * fft_size`.

The `twiddle` table is passed in the argument `twiddle_table`, which must contain at least `fft_size` twiddle factors for `rfft2d_fr16` and at least `3*fft_size/4` twiddle factors for `rfft2d_fr32`. The table should be initialized with complex twiddle factors in which the real coefficients are positive cosine values and the imaginary coefficients are negative sine values. The function `twidfft2d_fr16` may be used to initialize the arrays for `rfft2d_fr16` and `rfft2d_fx_fr16`, while `twidfft2d_fr32` may be used to initialize the arrays for `rfft2d_fr32` and `rfft2d_fx_fr32`.

If the `twiddle` table has been generated for an `fft_size` FFT, the `twiddle_stride` argument should be set 1. On the other hand, if the `twiddle` table has been generated for an FFT of size x, where `x>fft_size`, then the `twiddle_stride` argument should be set to `x/fft_size`. The `twiddle_stride` argument therefore allows the same `twiddle` table to be used for different sizes of FFT. (The `twiddle_stride` argument cannot be either zero or negative).

To avoid overflow, the functions scale the output by `fft_size*fft_size`.

The `rfft2d_fr16` arguments `block_exponent` and `scale_method` have been added for future expansion. These arguments are ignored by the function.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The `rfft2d` functions abort if the twiddle stride is less than 1, or if `fft_size` is less than 4 for `rfft2d_fr16` or `rfft2d_fx_fr16`, or if `fft_size` is less than 16 for `rfft2d_fr32` or `rfft2d_fx_fr32`.

Algorithm

The following equation is the basis of the algorithm.

$$c(i, j) = \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} a(k, l) \cdot e^{(-2\pi \cdot (i \cdot k + j \cdot l))/n}$$

where:

i = {0, 1, ..., n-1}

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

Domain

The argument `fft_size` must be a power of 2 and at least 4 for `rfft2d_fr16` and `rfft2d_fx_fr16`, and at least 16 for `rfft2d_fr32` and `rfft2d_fx_fr32`.

Example

```
#include <filter.h>

#define FFT_SIZE1         128
#define FFT_SIZE2          32
#define TWIDDLE_STRIDE1  (FFT_SIZE1 / FFT_SIZE1)
#define TWIDDLE_STRIDE2  (FFT_SIZE1 / FFT_SIZE2)

complex_fract32  out_a[FFT_SIZE1][FFT_SIZE1];
complex_fract32  out_b[FFT_SIZE2][FFT_SIZE2];
complex_fract32  in[FFT_SIZE2][FFT_SIZE2];
complex_fract32  tmp[FFT_SIZE1][FFT_SIZE1];
complex_fract32  twiddle[(3*FFT_SIZE1)/4];

fract32          *in1  = (fract32*)&out_a;
complex_fract32  *out1 = (complex_fract32*)&out_a;
fract32          *in2  = (fract32*)&in;
complex_fract32  *out2 = (complex_fract32*)&out_b;
complex_fract32  *tmp  = (complex_fract32*)&temp;
twidfft2d_fr32 (twiddle, FFT_SIZE1);
```

```
/* In-place computation */
rfft2d_fr32(in1, tmp, out1, twiddle, TWIDDLE_STRIDE1, FFT_SIZE1);
rfft2d_fr32(in2, tmp, out2, twiddle, TWIDDLE_STRIDE2, FFT_SIZE2);
```

rms

Root mean square

Synopsis

```
#include <stats.h>

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

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

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

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

fract32 rms_fr32(const fract32   samples[],
                  int      sample_length);

_Fract rms_fx16(const _Fract     samples[],
                 int       sample_length);

long _Fract rms_fx32(const long _Fract   samples[],
                      int          sample_length);
```

Description

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2
> Utility ROM for further information.

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`

Domain

[-3.4e38 , +3.4e38]	for `rmsf( )`
[-1.7e308 , +1.7e308]	for `rmsd( )`
[-1.0 , +1.0)	for `rms_fr16( )`, `rms_fx16( )`, `rms_fr32( )`, and `rms_fx32( )`

rsqrt

Reciprocal square root

Synopsis

```
#include <math.h>

float rsqrtf (float a);
double rsqrt (double a);
long double rsqrtd (long double a);
```

Description

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

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

The following equation is the basis of the algorithm.

$$c = \frac{1}{\sqrt{a}}$$

Domain

[0.0 , 3.4e38]	for `rsqrtf( )`
[0.0 , +1.7e308]	for `rsqrtd( )`

twidfftrad2

Generate FFT twiddle factors for radix-2 FFT

Synopsis

```
#include <filter.h>

void twidfftrad2_fr16(complex_fract16  twiddle_table[],
                      int              fft_size);
void twidfftrad2_fr32(complex_fract32  twiddle_table[],
                      int              fft_size);
```

Description

The `twidfftrad2` functions calculate complex twiddle coefficients for an FFT of size `fft_size` and return the coefficients in the vector `twiddle_table`. The size of the vector, which is known as a *twiddle table*, must be at least `fft_size`/2. It contains pairs of sine and cosine values that are used by an FFT function to calculate a Fast Fourier Transform. The table generated by the function twidfftrad2_fr16 may be used by any of the functions `cfft_fr16`, `ifft_fr16`, `rfft_fr16` and `rfft_fx_fr16`, and the table generated by the function `twidfftrad2_fr32` may be used by any of the functions `cfft_fr32`, `ifft_fr32`, `rfft_fr32` and `rfft_fx_fr32`.

A twiddle table of a given size contains constant values, and so typically such a table would be generated only once during the development cycle of an application and would thereafter be preserved by the application in some suitable form.

An application that calculates FFTs of different sizes does not require multiple twiddle tables. A single twiddle table can be used to compute the FFTs provided that the table is created for the largest FFT that the application expects to generate. Each of the FFT functions cfft, ifft, and rfft have a twiddle stride argument that the application would set to 1 when it is generating an FFT with the largest number of data points.

To generate smaller FFTs, the twiddle stride argument should be set according to the formula:

$$\frac{largest\,FFT\,size}{current\,FFT\,size}$$

For example, if a twiddle table had been created for a 1024-point FFT, then the same table could also be used to calculate a 256-point FFT by setting the twiddle stride argument to 4.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

These functions calculate a lookup table of complex twiddle factors. The coefficients generated are:

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

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

where:

n = `fft_size`

k = {0, 1, 2, ..., n/2 - 1}

Domain

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

Example

```
#include <filter.h>

#define FFT_SIZE1  256
#define FFT_SIZE2  64
#define TWID_SIZE  (FFT_SIZE1/2)

complex_fract32  input1[FFT_SIZE1];
complex_fract32  output1[FFT_SIZE1];
complex_fract32  input2[FFT_SIZE2];
complex_fract32  output2[FFT_SIZE2];
complex_fract32  twiddles[TWID_SIZE];
int  block_exponent1, block_exponent2;
int  scale_method = 1;

twidfftrad2_fr32 (twiddles, FFT_SIZE1);

cfft_fr32 (input1, output1, twiddles, 1, FFT_SIZE1,
          &block_exponent1, scale_method);

cfft_fr32 (input1, output2, twiddles, (FFT_SIZE1/FFT_SIZE2),
          FFT_SIZE2, &block_exponent2, scale_method);
```

twidfftrad2_fr16_8k_table

Synopsis

```
#include <filter.h>

const complex_fract16 twidfftrad2_fr16_8k_table[4096];
```

Description

The array `twidfftrad2_fr16_8k_table` is a pre-computed twiddle table that can be used to compute an FFT of 4096 points using the set of `fract16 radix-2` FFT functions (i.e. `cfft_fr16`, `ifft_fr16`, `rfft_fr16`, or `rfft_fx_fr16`).

The pre-computed twiddle table can be used to compute FFTs whose size is smaller than 4K by setting the twiddle stride argument of the FFT function to (`4096/fftsize`) where `fftsize` is the size of the smaller FFT.

For the ADSP-BF7xx family parts, a copy of the twiddle table is also available in the platform's L2 ROM. By default, references to the table by applications that are built for these parts are resolved to the ROM copy unless either use of the ROM is not compatible with the project settings or if the switch `-no-utility-rom` is specified.

The memory that twiddle data is linked to has an impact on the performance of the FFT functions. Best performance is achieved by using a twiddle table that is linked into L1 memory. Note that the ADSP-BF7xx ROM definition uses L2 memory so the performance achievable using the ROM may not be suitable for some applications.

Example

```
#include <filter.h>

#define FFT_SIZE_8K (8*1024)
#define FFT_SIZE_2K (2*1024)

void do_8k_rad2_cfft_fr16(complex_fract16 input[],
                          complex_fract16 output[])
{
   const int twiddle_stride = 1;
   const int fft_size = FFT_SIZE_8K;

   cfft_fr16(input, output,
             twidfftrad2_fr16_8k_table,
             twiddle_stride, fft_size);
}

void do_2k_rad2_rfft_fr16(fract16 *input,
                          complex_fract16 *output)
{
   const int twiddle_stride = FFT_SIZE_8K/FFT_SIZE_2K;
   const int fft_size = FFT_SIZE_2K;

   rfft_fr16(input, output,
```

```
            twidfftrad2_fr16_8k_table,
            twiddle_stride, fft_size);
}
```

twidfftrad2_fr32_4k_table

Synopsis

Pre-calculated FFT twiddle data array.

```
#include <filter.h>

const complex_fract32 twidfftrad2_fr32_4k_table[2048];
```

Description

The array `twidfftrad2_fr32_4k_table` is a pre-computed twiddle table that can be used to compute an FFT of 4096 points using the set of `fract32 radix-2` FFT functions (i.e. `cfft_fr32`, `ifft_fr32`, `rfft_fr32`, or `rfft_fx_fr32`).

The pre-computed twiddle table can be used to compute FFTs whose size is smaller than 4K by setting the twiddle stride argument of the FFT function to (`4096/fftsize`) where `fftsize` is the size of the smaller FFT.

For the ADSP-BF7xx family parts, a copy of the twiddle table is also available in the platform's L2 ROM. By default, references to the table by applications that are built for these parts are resolved to the ROM copy unless either use of the ROM is not compatible with the project settings or if the switch `-no-utility-rom` is specified.

The memory that twiddle data is linked to will have an impact on the performance of the FFT functions. Best performance will be achieved by using a twiddle table that is linked into L1 memory. Note that the ADSP-BF7xx ROM definition uses L2 memory so the performance achievable using the ROM may not be suitable for some applications.

Example

```
#include <filter.h>

#define FFT_SIZE_4K (4*1024)
#define FFT_SIZE_2K (2*1024)

void do_4k_rad2_cfft_fr32(complex_fract32 input[],
                          complex_fract32 output[])
{
    const int twiddle_stride = 1;
    const int fft_size = FFT_SIZE_4K;

    cfft_fr32(input, output,
            twidfftrad2_fr32_4k_table,
            twiddle_stride, fft_size);
}
```

```
void do_2k_rad2_rfft_fr32(fract32 *input,
                          complex_fract32 *output)
{
   const int twiddle_stride = FFT_SIZE_4K/FFT_SIZE_2K;
   const int fft_size = FFT_SIZE_2K;

   rfft_fr32(input, output,
           twidfftrad2_fr32_4k_table,
           twiddle_stride, fft_size);
}
```

twidfftf

Generate FFT twiddle factors for a fast FFT

Synopsis

```
#include <filter.h>

void twidfftf_fr16(complex_fract16 twiddle_table[ ],
                   int             fft_size);

void twidfftf_fr32(complex_fract32 twiddle_table[ ],
                   int             fft_size);
```

Description

The `twidfftf` functions generate complex twiddle factors for the fast mixed-radix `cfftf`, `ifftf`, and `rfftf` functions. The twiddle factors are pairs of cosine and sine values that are stored in the vector twiddle_table; the FFT functions will then use this table to generate a Fast Fourier Transform. The size of the twiddle table must be at least `3*fft_size/4` where `fft_size` is the number of points in the FFT. The table generated by the function `twidfftf_fr16` may be used by any of the functions `cfftf_fr16`, `ifftf_fr16`, `rfftf_fr16`, and `rfftf_fx_fr16`, and the table generated by the function `twidfftf_fr32` can be used by any of the functions `cfftf_fr32`, `ifftf_fr32`, `rfftf_fr32`, and `rfftf_fx_fr32`.

A twiddle table of a given size contains constant values, and so typically such a table would be generated only once during the development cycle of an application and would thereafter be preserved by the application in some suitable form.

An application that calculates FFTs of different sizes does not require multiple twiddle tables. A single twiddle table can be used to compute the FFTs provided that the table is created for the largest FFT that the application expects to generate. Each FFT function 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 smaller FFTs, the twiddle stride argument should be set according to the formula:

$$\frac{largest\,FFT\,size}{current\,FFT\,size}$$

For example, if a twiddle table had been created for a 1024-point FFT, then the same table could also be used to calculate a 256-point FFT by setting the twiddle stride argument to 4.

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

None

Algorithm

The functions calculate a lookup table of complex twiddle factors. The coefficients generated are:

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

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

where:

n = fft_size

k = {0, 1, 2, ..., n - 1}

Domain

The number of points in the FFT, fft_size, must be a power of 2. It must also be at least 8 for the >cfftf and ifftf functions, and must be at least 16 for the rfftf functions.

Example

```
#include <filter.h>
#define FFT_SIZE1        256
#define FFT_SIZE2         64
#define TWIDDLE_SIZE  ((3*FFT_SIZE1)/4)

complex_fract32  in1[FFT_SIZE1];
complex_fract32  out1[FFT_SIZE1];
complex_fract32  in2[FFT_SIZE2];
complex_fract32  out2[FFT_SIZE2];
complex_fract32  twiddles[TWIDDLE_SIZE];

twidfftf_fr32 (twiddles, FFT_SIZE1);
```

```
cfftf_fr32(in1, out1, twiddles, 1, FFT_SIZE1);

cfftf_fr32(in2, out2, twiddles, FFT_SIZE1/FFT_SIZE2, FFT_SIZE2);
```

twidfftf_fr16_8k_table

Synopsis

Pre-calculated FFT twiddle data array.

```
#include <filter.h>

const complex_fract16  twidfftf_fr16_8k_table[6144];
```

Description

The array `twidfftf_fr16_8k_table` is a pre-computed twiddle table that can be used to compute an FFT of 8192 points using the set of `fract16` fast `radix-4` FFT functions (i.e. `cfftf_fr16`, `ifftf_fr16`, `rfftf_fx_fr16`).

The pre-computed twiddle table can be used to compute FFTs whose size is smaller than 8K by setting the twiddle stride argument of the FFT function to (`8192/fftsize`) where `fftsize` is the size of the smaller FFT.

For the ADSP-BF7xx processors, a copy of the twiddle table is also available in the platform's L2 ROM. By default, references to the table by applications that are built for these parts are resolved to the ROM copy unless either use of the ROM is not compatible with the project settings or if the switch `-no-utility-rom` has been specified.

The memory that twiddle data is linked to will have an impact on the performance of the FFT functions. Best performance is achieved by using a twiddle table that is linked into L1 memory. Note that the ADSP-BF7xx ROM definition uses L2 memory so the performance achievable using the ROM may not be suitable for some applications.

A twiddle table to compute a `radix-4` FFT can also be used to compute an FFT using the set of `fract16` `radix-2` FFT functions (i.e. `cfft_fr16`, `ifft_fr16`, `rfft_fx_fr16`). This is possible because the first N samples in a twiddle table for the `radix-2` FFT functions are identical to the first N samples in a twiddle table for a `radix-4` FFT.

Example

```
#include <filter.h>

#define FFT_SIZE_8K (8*1024)
#define FFT_SIZE_4K (4*1024)

void do_8k_cfftf_fr16(complex_fract16 input[], complex_fract16 output[])
{
   const int twiddle_stride = 1;
   const int fft_size = FFT_SIZE_8K;
```

```
    cfftf_fr16(input, output,
              twidfftf_fr16_8k_table,
              twiddle_stride, fft_size);
}


void do_4k_rfftf_fr16(fract16 input[], complex_fract16 output[])
{
    const int twiddle_stride = FFT_SIZE_8K/FFT_SIZE_4K;
    const int fft_size = FFT_SIZE_4K;

    rfftf_fr16(input, output,
              twidfftf_fr16_8k_table, twiddle_stride,
              fft_size);
}
```

twidfftf_fr32_4k_table

Synopsis

Pre-calculated FFT twiddle data array.

```
#include <filter.h>

const complex_fract32 twidfftf_fr32_4k_table[3072];
```

Description

The array `twidfftf_fr32_4k_table` is a pre-computed twiddle table that can be used to compute an FFT of 4096 points using the set of `fract32` 2-D FFT functions (i.e. `cfft2d_fr32`, `ifft2d_fr32`, `rfft2d_fr32`) or the set of `fract32` fast mixed-radix FFT functions (i.e. `cfftf_fr32`, `ifftf_fr32`, `rfftf_fx_fr32`).

The pre-computed twiddle table can be used to compute FFTs whose size is smaller than 4K by setting the twiddle stride argument of the FFT function to (`4096/fftsize`) where `fftsize` is the size of the smaller FFT.

For the ADSP-BF7xx processors, a copy of the twiddle table is also available in the platform's L2 ROM. By default, references to the table by applications that are built for these parts are resolved to the ROM copy unless either use of the ROM is not compatible with the project settings or if the switch `-no-utility-rom` has been specified.

The memory that twiddle data is linked to has an impact on the performance of the FFT functions. Best performance is achieved by using a twiddle table that is linked into L1 memory. Note that the ADSP-BF7xx ROM definition uses L2 memory so the performance achievable using the ROM may not be suitable for some applications.

A twiddle table to compute a mixed-radix FFT can also be used to compute an FFT using the set of `fract32` `radix-2` FFT functions (i.e. `cfft_fr32`, `ifft_fr32`, `rfft_fr32`). This is possible because the first N samples in a twiddle table for the `radix-2` FFT functions are identical to the 1st N samples in a twiddle table for a mixed-radix FFT.

Example

```
#include <filter.h>

#define FFT_SIZE_4K (4*1024)
#define FFT_SIZE_2K (2*1024)

void do_4k_cfftf_fr32(complex_fract32 input[], complex_fract32 output[])
{
   const int twiddle_stride = 1;
   const int fft_size = FFT_SIZE_4K;

   cfftf_fr32(input, output,
            twidfftf_fr32_4k_table,
            twiddle_stride, fft_size);
}

void do_2k_rfftf_fr32(fract32 *input, complex_fract32 *output)
{
   const int twiddle_stride = FFT_SIZE_4K/FFT_SIZE_2K;
   const int fft_size = FFT_SIZE_2K;

   rfftf_fr32(input, output, twidfftf_fr32_4k_table,
            twiddle_stride, fft_size);
}
```

twidfft2d

Generate FFT twiddle factors for 2-D FFT

Synopsis

```
#include <filter.h>

void twidfft2d_fr16 (complex_fract16  twiddle_table[],
                     int              fft_size);
void twidfft2d_fr32 (complex_fract32  twiddle_table[],
                     int              fft_size);
```

Description

The `twidfft2d` functions calculate complex twiddle coefficients for a 2-D FFT of size `fft_size` and return the coefficients in the vector `twiddle_table`. The size of the vector, which is known as a twiddle table, must be at least `fft_size` for `twidfft2d_fr16`, and at least `3*fft_size/4` for `twidfft2d_fr32`. It contains pairs of sine and cosine values that are used by an FFT function to calculate a Fast Fourier Transform. The table generated by the function `twidfft2d_fr16` can be used by any of the functions `cfft2d_fr16`, `ifft2d_fr16`, `rfft2d_fr16`, and `rfft2d_fx_fr16`, and the table generated by the function

`twidfft2d_fr32` can be used by any of the functions `cfft2d_fr32`, `ifft2d_fr32`, `rfft2d_fr32`, and `rfft2d_fx_fr32`.

A twiddle table of a given size contains constant values, and so typically such a table would be generated only once during the development cycle of an application and would thereafter be preserved by the application in some suitable form.

An application that calculates FFTs of different sizes does not require multiple twiddle tables. A single twiddle table can be used to compute the FFTs provided that the table is created for the largest FFT that the application expects to generate. Each 2-D FFT function 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 smaller FFTs, the twiddle stride argument should be set according to the formula:

$$\frac{largest\,FFT\,size}{current\,FFT\,size}$$

For example, if a twiddle table had been created for a 1024-point FFT, then the same table could also be used to calculate a 256-point FFT by setting the twiddle stride argument to 4.

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

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

The samples generated are:

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

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

where:

n = `fft_size`

k = {0, 1, 2, ..., n-1}

Domain

The number of points in the FFT must be a power of 2, and must be at least 4 for `cfft2d_fr16`, `ifft2d_fr16`, `rfft2d_fr16`, and `rfft2d_fx_fr16`, at least 8 for `cfft2d_fr32` and `ifft2d_fr32` and at least 16 for the `rfft2d_fr32` and `rfft2d_fx_fr32` functions.

var

Variance

Synopsis

```
#include <stats.h>

float varf(const float  samples[ ], int sample_length);
double var(const double samples[ ], int sample_length);
long double vard(const long double samples[ ], int sample_length);
fract16 var_fr16(const fract16 samples[ ], int sample_length);
_Fract var_fx16(const _Fract  samples[ ],
                        int    sample_length);
fract32 var_fr32(const fract32  samples[ ],
                        int       sample_length);
long _Fract var_fx32(const long _Fract samples[ ],
                            int         sample_length);
```

Description

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

NOTE: These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Error Conditions

The `var_fr16` and `var_fx16` functions can be used to compute the variance of up to 65535 input data with a value of 0x8000 before the sum a_i saturates. The `var_fr32` and `var_fx32` functions can be used to compute the variance of up to 4294967295 input data with a value of 0x80000000 before the sum a_i saturates.

Algorithm

The following equation is the basis of the algorithm.

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

where:

a = `samples`

n = `sample_length`

Domain

[-3.4e38 , +3.4e38]	for `varf( )`
[-1.7e308 , +1.7e308]	for `vard( )`
[-1.0 , +1.0)	for `var_fr16( )`, `var_fx16( )`, `var_fr32( )`, `var_fx32( )`

zero_cross

Count zero crossings

Synopsis

```
#include <stats.h>

int zero_crossf (const float   samples[ ],
                        int     samples_length);
int zero_cross  (const double  samples[ ],
                        int     samples_length);
int zero_crossd (const long double  samples[ ],
                        int         samples_length);
int zero_cross_fr16 (const fract16  samples[ ],
                            int      samples_length);
int zero_cross_fx16 (const _Fract   samples[ ],
                             int     samples_length);
int zero_cross_fr32 (const fract32  samples[ ],
                            int      samples_length);
int zero_cross_fx32 (const long _Fract samples[ ],
                            int          samples_length);
```

Description

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

> **NOTE:** These functions are included in ROM for the ADSP-BF7xx processors. See ADSP-BF7xx Processors L2 Utility ROM for further information.

Algorithm

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

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

the number of zeros is increased by one.

Domain

[-3.4e38 , +3.4e38]	for `zero_crossf( )`
[-1.7e308 , +1.7e308]	for `zero_crossd( )`
[-1.0 , +1.0)	for `zero_cross_fr16( )`, `zero_cross_fx16( )`, `zero_cross_fr32( )`, `zero_cross_fx32( )`

6 Multicore Programming

The Blackfin family of processors includes dual-core parts, such as ADSP-BF561 and ADSP-BF609. In addition to other features, dual-core processors add a new dimension to application development. The dual-core nature of the processor presents additional challenges to the programmer - this section addresses these challenges within the context of CrossCore Embedded Studio.

NOTE: The documentation for the ADSP-BF561 processor uses "core A" and "core B" nomenclature, while the documentation for the ADSP-BF609 processor uses "core 0" and "core 1", respectively. This chapter uses the latter.

The appendix contains:

- Dual-Core Blackfin Architecture Overview

- Application Model

- Compiler and Library Support

Dual-Core Blackfin Architecture Overview

Each dual-core Blackfin processor has two Blackfin cores (core 0 and core 1), each with its own internal L1 memory. There is a common internal memory shared between the two cores, and both cores share access to external memory.

Each core functions independently: they have their own reset address, event vector table, instruction and data caches, and more. On reset, core 0 starts running from its reset address, while core 1 is disabled. Core 1 starts running when it is enabled by core 0.

NOTE: CCES enables core 1 when it connects to an EZ-KIT®, as part of the program download process.

When core 1 starts running, it runs its own application from its own reset address.

Applications running on the two cores can use the `TESTSET` instruction to serialize access to shared resources. The `TESTSET` instruction reads and updates a memory location in an atomic fashion. Applications and libraries can build semaphores and other synchronization mechanisms from this primitive.

Refer to the appropriate hardware reference manual for detailed information.

Application Model

Analog Devices recommends the following application model:

- Each core is treated as a separate processor, running its own program (meaning you build a separate `.dxe` image for each processor cores).

- Communication and synchronization between the cores is managed via an appropriate communication protocol; CCES includes an implementation of the MCAPI protocol for this purpose.

- Physically shared memory locations - memory accessible by both cores - should be allocated to one core's memory map or the other. The same locations should not be mapped into both cores' memory spaces, with the exception being those locations required to implement the communications protocol.

- Where the applications on each core need to access the same data, the communications protocol is used.

- No code is shared between the applications.

This model provides for a predictable development environment, and allows you to choose different implementation approaches on each core. For example, you can use an RTOS on core 0 and managing several control threads, while core 1 is not using an RTOS, instead repeatedly reading packets of data from one peripheral, processing them, and writing the results to another peripheral.

Compiler and Library Support

CCES provides some low-level support for multicore programming, which are described in this section. Higher-level support also is available, documented elsewhere.

Project Creation

When you create new projects for multicore processors, the IDE's default behavior is to create a project for each core, with the following conveniences:

- Template code for the `main()` function for each core. The `main()` function for core 0 includes a call of `adi_core_1_enable()`, to start core 1 running.

- A generated `.ldf` file for each core, partitioning physically-shared memory between the cores, and reserving some space for the inter-core communication protocol (the MCAPI implementation provided with CCES employs this space). You can choose not to have a generated `.ldf` file, if you prefer to use the default file, or provide your own.

- Generated startup code for each core, with memory initialization where necessary. You can choose not to have generated startup code, if you prefer to use the default, or provide your own.

When you create your project, you can also choose to add the MCAPI add-in for communication.

LDF Files

If you allow the IDE to provide you with generated `.ldf` files for your projects, each `.ldf` file is configured according to the corresponding core.

If you elect to use the default `.ldf` file, instead of having a generated one, the default `.ldf` is used for both cores. The default `.ldf` file respects two link-time macros:

- If the CORE0 macro is defined, the `.ldf` file defines memory spaces for core 0.

- If the CORE1 macro is defined, the `.ldf` file defines memory spaces for core 1.

- If neither macro is defined, the `.ldf` file behaves as though CORE0 is defined.

- If both macros are defined, the `.ldf` file raises an error message.

Projects created for core 0 do not define either macro, so they use the default behavior of building for core 0, while projects created for core 1 define the CORE1 macro. If you choose to create your own projects that use the default `.ldf` file, ensure you define the appropriate COREx macro at link-time.

Startup Code

If you allow the IDE to create projects with generated startup code, each project is created with an appropriate startup file, configured for the appropriate core. The startup code for core 1 ensures that the application on core 0 has released core 1, before proceeding with the initialization. This prevents core 1's initialization happening too soon, when the core has been released by the emulator during the loading process.

If you elect to use the default startup code, the same startup code is used for both cores; it identifies the executing core at run-time, and ensures that core 1 does not initialize too soon.

MCAPI

CCES includes an implementation of the Multicore Communications API (MCAPI) protocol. For more information, refer to the *Multicore Communications API (MCAPI) Specification* documentation, which can be found in the CCES online help.

Library Functions

The run-time library provides several low-level functions for multicore processors:

- Releasing core 1. The `adi_core_1_enable()` function allows core 1 to start running and informs core 1 that the release command comes from the application running on core 0, and not from the emulator. Where the processor supports it, `adi_core_1_disable()` is also available. For more information, see adi_core_enable, adi_core_1_enable, adi_core_1_disable, adi_core_b_enable, adi_core_enable, adi_core_1_enable, adi_core_1_disable, adi_core_b_enable, adi_core_enable, adi_core_1_enable, adi_core_1_disable, adi_core_b_enable in the *C/C++ Run-Time Library* chapter.

- Identifying the number of cores in the system. The compiler defines the __NUM_CORES__ macro, for all processors. This macro has a value greater than one for all multicore processors.

- Identifying the current core. The `adi_core_id()` function returns the numeric identifier of the calling core. For more information, see adi_core_id in the *C/C++ Run-Time Library* chapter.

- Acquiring spinlocks. In general, Analog Devices recommends that a higher-level protocol, such as MCAPI, is used for inter-core communications. However, low-level functions are available. For more information, see adi_acquire_lock, adi_try_lock, adi_release_lock, adi_acquire_lock, adi_try_lock, adi_release_lock, adi_acquire_lock, adi_try_lock, adi_release_lock in the *C/C++ Run-Time Library* chapter.

7 Utility ROMs

ADSP-BF7xx Processors L2 Utility ROM

The ADSP-BF7xx processors have L2 ROM, containing run-time library functions, device-drivers, and FFT twiddle tables. CCES supports these ROM components.

Support for Controlling Utility ROM

Support for the ADSP-BF7xx utility ROM `libc`, `libcc`, `libdsp`, and `libdrv` functionality is enabled by default when building applications for the ADSP-BF7xx parts and silicon revisions other than `none` or `any`.

The utility ROM support is enabled by the symbols (for the utility ROM contents) being defined in the LDF when linking. Some or all of these symbols can be excluded with the definitions of various `NO_UTILITY_ROM` linker disable macros. Excluding a ROM symbol with a disable macro causes any references of excluded functions or data to use definitions from archive (`.dlb`) libraries that are linked into RAM.

There are two main compiler switches to enable or disable the entire L2 Utility ROM contents. These switches control the definition of the global disable macro `NO_UTILITY_ROM`:

- -utility-rom

- -no-utility-rom

These are available in project settings under *CrossCore Blackfin Linker: Preprocessor: Processor: Utility Rom.*

The run-time library and device drivers components of the utility ROM also are under independent lower-level macro control and can be disabled by defining the following macros when linking (using *CrossCore Blackfin Linker: Preprocessor: Preprocessor definitions (-MD)*):

- `NO_UTILITY_ROM_LIBDSP` - use `libdsp` inputs from archive `dlb` files in RAM instead of those available in ROM

- `NO_UTILITY_ROM_LIBCC` - use `libcc` inputs from archive `dlb` files in RAM instead of those available in ROM

- `NO_UTILITY_ROM_LIBC` - use `libc` inputs from archive `dlb` files in RAM instead of those available in ROM

- `NO_UTILITY_ROM_LIBDRV` - use `libdrv` inputs from archive `dlb` files in RAM instead of those available in ROM. You also must define `NO_UTILITY_ROM_LIBDRV` as a compile-time macro (using *CrossCore Blackfin C/C++ Compiler: Preprocessor: Preprocessor definitions (-D)* to exclude initialization code in the generated project file, `adi_initialize.c`.

Each individual utility ROM function that is part of the `libc`, `libcc`, `libdsp` run-time libraries and the `libdrv` device drivers can also be disabled. The macros corresponding to each Utility ROM function are included in the following tables.

Performance Considerations

L2 ROM can be slower to access and execute than L1 RAM, so it is necessary to consider disabling ROM versions of functions or FFT twiddle data that are performance critical to the end application.

L2 RAM Reserved for ROM

Some of the ROM components require addresses in RAM to be reserved when linking as those addresses are used by the ROM. The addresses reserved are in L2 RAM, and the CCES LDFs are setup to do the required reserves automatically.

Table 7-1: L2 RAM Reserved for ROM

Rom Component	Reserved Start	Reserved End
Device drivers	0x0800_2000	0x0800_2003

There is no RAM reserved for any of the `libc`, `libcc`, `libdsp`, and FFT `twiddle` tables in the utility ROM.

Utility ROM Contents

The ADSP-BF7xx processors include extensive functionality in L2 ROM space, which includes:

- Commonly-accessed run-time library support, including DSP library functions and math emulation.

  A list of the functions and data exported by the L2 utility ROM is included later in this document.

- FFT twiddle tables.

  The `filter.h` pre-computed twiddle tables are defined in the utility ROM. The memory that twiddle data is linked to has an impact on the performance of the FFT functions. Best performance is achieved by using a twiddle table that is linked into L1 memory. Note that the ADSP-BF7xx utility ROM is in L2 memory so the performance achievable using the ROM may not be suitable for some applications.

Table 7-2: Utility ROM twiddle Tables

Symbol	Disable Macro	Documentation Link
twidfftf_fr16_8k_table	NO_UTILITY_ROM_twidfftf_fr16_8k_table	See twidfftf_fr16_8k_table
twidfftf_fr32_4k_table	NO_UTILITY_ROM_twidfftf_fr32_4k_table	See twidfftf_fr32_4k_table
twidfftrad2_fr16_8k_table	NO_UTILITY_ROM_twidfftrad2_fr16_8k_table	See twidfftrad2_fr16_8k_table
twidfftrad2_fr32_4k_table	NO_UTILITY_ROM_twidfftrad2_fr32_4k_table	See twidfftrad2_fr32_4k_table

- Device drivers.

  The Rotary Counter, HADC (revision 1.0 only), RSI (revision 1.0 only), PPI, SPI SPORT, TWI, and UART drivers are provided in the utility ROM.

Table 7-3: Utility Rom Device Drivers

Driver	Disable Macro
Rotary Counter	NO_UTILITY_ROM_COUNTER
HADC	NO_UTILITY_ROM_HADC
PPI	NO_UTILITY_ROM_PPI
RSI	NO_UTILITY_ROM_RSI
SPI	NO_UTILITY_ROM_SPI
SPORT	NO_UTILITY_ROM_SPORT
TWI	NO_UTILITY_ROM_TWI
UART	NO_UTILITY_ROM_UART

NOTE: Adding a driver to an application as source via the add-in mechanism only disables the ROM version of that specific driver. Any API calls to other drivers that are including in the ROM still uses the ROM.

Utility ROM Simulation

The ADSP-BF7xx simulator sessions and chipfactory command-line simulator select and include the ROM DXE files required for the application DXE being loaded. The simulators load the selected ROM DXE files as prerequisites to the application to simulate correctly any ROM used in the application.

Utility ROM `libc` Functions

Selected `math.h`, `string.h`, `ctype.h`, and `stdlib.h` functions are included in the utility ROM.

Table 7-4: Functions from C Run-Time Library Defined in Utility ROM

Function	Header	Disable Macro
abs	stdlib.h	NO_UTILITY_ROM_abs
acosd	math.h	NO_UTILITY_ROM__acosd
acosf	math.h	NO_UTILITY_ROM__acosf
asind	math.h	NO_UTILITY_ROM__asind
asinf	math.h	NO_UTILITY_ROM__asinf
atan2d	math.h	NO_UTILITY_ROM__atan2d
atan2f	math.h	NO_UTILITY_ROM__atan2f
atand	math.h	NO_UTILITY_ROM__atand
atanf	math.h	NO_UTILITY_ROM__atanf
ceild	math.h	NO_UTILITY_ROM__ceild
ceilf	math.h	NO_UTILITY_ROM__ceilf
cosd	math.h	NO_UTILITY_ROM__cosd
cosf	math.h	NO_UTILITY_ROM__cosf
coshd	math.h	NO_UTILITY_ROM__coshd
coshf	math.h	NO_UTILITY_ROM__coshf
expd	math.h	NO_UTILITY_ROM__expd
expf	math.h	NO_UTILITY_ROM__expf
fabs	math.h	NO_UTILITY_ROM_fabs
fabsd	math.h	NO_UTILITY_ROM__fabsd
fabsf	math.h	NO_UTILITY_ROM_fabsf
floord	math.h	NO_UTILITY_ROM__floord
floorf	math.h	NO_UTILITY_ROM__floorf
fmaxd	math.h	NO_UTILITY_ROM__fmaxd
fmind	math.h	NO_UTILITY_ROM__fmind
fmodf	math.h	NO_UTILITY_ROM__fmodf
frexpd	math.h	NO_UTILITY_ROM__frexpd
frexpf	math.h	NO_UTILITY_ROM__frexpf
isalnum	ctype.h	NO_UTILITY_ROM_isalnum
isalpha	ctype.h	NO_UTILITY_ROM_isalpha

Table 7-4: Functions from C Run-Time Library Defined in Utility ROM (Continued)

Function	Header	Disable Macro
iscntrl	ctype.h	NO_UTILITY_ROM_iscntrl
isdigit	ctype.h	NO_UTILITY_ROM_isdigit
isgraph	ctype.h	NO_UTILITY_ROM_isgraph
isinf	math.h	NO_UTILITY_ROM_isinf
isinfd	math.h	NO_UTILITY_ROM_isinfd
islower	ctype.h	NO_UTILITY_ROM_islower
isnan	math.h	NO_UTILITY_ROM_isnan
isnand	math.h	NO_UTILITY_ROM_isnand
isprint	ctype.h	NO_UTILITY_ROM_isprint
ispunct	ctype.h	NO_UTILITY_ROM_ispunct
isspace	ctype.h	NO_UTILITY_ROM_isspace
isupper	ctype.h	NO_UTILITY_ROM_isupper
isxdigit	ctype.h	NO_UTILITY_ROM_isxdigit
labs	stdlib.h	NO_UTILITY_ROM_labs
ldexpd	math.h	NO_UTILITY_ROM__ldexpd
llabs	stdlib.h	NO_UTILITY_ROM_llabs
lldiv	stdlib.h	NO_UTILITY_ROM_lldiv
llmax	math.h	NO_UTILITY_ROM__llmax
llmin	math.h	NO_UTILITY_ROM__llmin
log10d	math.h	NO_UTILITY_ROM__log10d
log10f	math.h	NO_UTILITY_ROM__log10f
logd	math.h	NO_UTILITY_ROM__logd
logf	math.h	NO_UTILITY_ROM__logf
max	math.h	NO_UTILITY_ROM__max
memchr	string.h	NO_UTILITY_ROM_memchr
memcmp	string.h	NO_UTILITY_ROM_memcmp
memcpy	string.h	NO_UTILITY_ROM_memcpy
memmove	string.h	NO_UTILITY_ROM_memmove
memset	string.h	NO_UTILITY_ROM_memset
min	math.h	NO_UTILITY_ROM__min
modfd	math.h	NO_UTILITY_ROM__modfd
modff	math.h	NO_UTILITY_ROM__modff

Table 7-4: Functions from C Run-Time Library Defined in Utility ROM (Continued)

Function	Header	Disable Macro
powd	`math.h`	`NO_UTILITY_ROM__powd`
powf	`math.h`	`NO_UTILITY_ROM__powf`
sind	`math.h`	`NO_UTILITY_ROM__sind`
sinf	`math.h`	`NO_UTILITY_ROM__sinf`
sinhd	`math.h`	`NO_UTILITY_ROM__sinhd`
sinhf	`math.h`	`NO_UTILITY_ROM__sinhf`
sqrtd	`math.h`	`NO_UTILITY_ROM__sqrtd`
sqrtf	`math.h`	`NO_UTILITY_ROM__sqrtf`
strcat	`string.h`	`NO_UTILITY_ROM_strcat`
strchr	`string.h`	`NO_UTILITY_ROM_strchr`
strcmp	`string.h`	`NO_UTILITY_ROM_strcmp`
strcoll	`string.h`	`NO_UTILITY_ROM_strcoll`
strcpy	`string.h`	`NO_UTILITY_ROM_strcpy`
strcspn	`string.h`	`NO_UTILITY_ROM_strcspn`
strftime	`time.h`	`NO_UTILITY_ROM_strftime`
strlen	`string.h`	`NO_UTILITY_ROM_strlen`
strncat	`string.h`	`NO_UTILITY_ROM_strncat`
strncmp	`string.h`	`NO_UTILITY_ROM_strncmp`
strncpy	`string.h`	`NO_UTILITY_ROM_strncpy`
strpbrk	`string.h`	`NO_UTILITY_ROM_strpbrk`
strrchr	`string.h`	`NO_UTILITY_ROM_strrchr`
strspn	`string.h`	`NO_UTILITY_ROM_strspn`
strstr	`string.h`	`NO_UTILITY_ROM_strstr`
strxfrm	`string.h`	`NO_UTILITY_ROM_strxfrm`
tand	`math.h`	`NO_UTILITY_ROM__tand`
tanf	`math.h`	`NO_UTILITY_ROM__tanf`
tanhd	`math.h`	`NO_UTILITY_ROM__tanhd`
tanhf	`math.h`	`NO_UTILITY_ROM__tanhf`
tolower	`ctype.h`	`NO_UTILITY_ROM_tolower`
toupper	`ctype.h`	`NO_UTILITY_ROM_toupper`

Utility ROM libcc Functions

Contents of the compiler support library, `libcc`, are included in the utility ROM with the exception of two run-time checking related functions (that use non-static data) and the definition of `_adi_stack_overflowed` (that requires an automatic breakpoint).

Table 7-5: Utility ROM `libcc` Contents

Compiler Support	Disable Macro
Support for division by multiplication of `signed short`	`NO_UTILITY_ROM___divmult_s16`
Support for division by multiplication of `signed int`	`NO_UTILITY_ROM___divmult_s32`
Support for division by multiplication of `unsigned short`	`NO_UTILITY_ROM___divmult_u16`
Support for division by multiplication of `unsigned int`	`NO_UTILITY_ROM___divmult_u32`
Memory copy	`NO_UTILITY_ROM___memcpy2`
Memory move	`NO_UTILITY_ROM___memmove2`
Double-precision floating-point addition	`NO_UTILITY_ROM___float64_add`
Double-precision floating-point addition	`NO_UTILITY_ROM___float64_add_inregs`
Arithmetic shift of `long long`	`NO_UTILITY_ROM___ashftli`
Conversion of `fract32` to `float`	`NO_UTILITY_ROM___fr32_to_float`
Conversion of `fract32` to `double`	`NO_UTILITY_ROM___fr32_to_float64`
Conversion of `fract32` to `double`	`NO_UTILITY_ROM___fr32_to_long_double`
`signed int` division	`NO_UTILITY_ROM___div32`
`signed int` division	`NO_UTILITY_ROM___div_s32`
`unsigned int` division	`NO_UTILITY_ROM___div_u32`
`unsigned int` division	`NO_UTILITY_ROM___udiv32`
`double` division	`NO_UTILITY_ROM___float64_div`
`signed 40-bit accum` division with rounding	`NO_UTILITY_ROM___divr_sk40`
`unsigned 40-bit accum` division with rounding	`NO_UTILITY_ROM___divr_uk40`
Rounding for 40-bit `accum` division	`NO_UTILITY_ROM___divr_k40_round`
`signed 32-bit integer` division and remainder	`NO_UTILITY_ROM___divrem_s32`
`signed int` division and remainder	`NO_UTILITY_ROM_div`
`signed long` division and remainder	`NO_UTILITY_ROM_ldiv`
`signed long long` division	`NO_UTILITY_ROM___divdi3`
`signed long long` division and remainder	`NO_UTILITY_ROM___divrem_s64`
`unsigned int` division and remainder	`NO_UTILITY_ROM___divrem_u32`
`unsigned long long` division and remainder	`NO_UTILITY_ROM___divrem_u64`

Table 7-5: Utility ROM `libcc` Contents (Continued)

Compiler Support	Disable Macro
`unsigned long long` division	`NO_UTILITY_ROM___udivdi3`
`unsigned` division support for fixed-point arithmetic	`NO_UTILITY_ROM___divrem_u96_u64`
`signed` 40-bit `accum` division with truncation	`NO_UTILITY_ROM___divt_sk40`
`unsigned` 40-bit `accum` division with truncation	`NO_UTILITY_ROM___divt_uk40`
Double-precision `float` comparison	`NO_UTILITY_ROM___float64_adi_eq`
Conversion from `float` to `double`	`NO_UTILITY_ROM___float32_to_float64`
Conversion from `float` to `int`	`NO_UTILITY_ROM___float32_to_int32_round_to_zero`
Conversion from `float` to `unsigned`	`NO_UTILITY_ROM___float32_to_unsigned_int32`
Conversion from `double` to `float`	`NO_UTILITY_ROM___float64_to_float32`
Conversion from `double` to `int`	`NO_UTILITY_ROM___float64_to_int32_round_to_zero`
Conversion from `double` to `long long`	`NO_UTILITY_ROM___float64_to_int64_round_to_zero`
Conversion from `double` to `unsigned int`	`NO_UTILITY_ROM___float64_to_unsigned_int32_round_to_zero`
Conversion from `double` to `unsigned long long`	`NO_UTILITY_ROM___float64_to_unsigned_int64_round_to_zero`
Conversion from `float` to `fract32`	`NO_UTILITY_ROM___float_to_fr32`
Conversion from `float` to `fract16`	`NO_UTILITY_ROM___float_to_fr16`
Conversion from `long long` to `float`	`NO_UTILITY_ROM___longlong64_to_float32`
Conversion from `unsigned long long` to `float`	`NO_UTILITY_ROM___unsigned_longlong64_to_float32`
`float` comparison	`NO_UTILITY_ROM___float32_adi_eq`
`float` comparison	`NO_UTILITY_ROM___float32_adi_gt`
`float` comparison	`NO_UTILITY_ROM___float32_adi_gteq`
`float` comparison	`NO_UTILITY_ROM___float32_adi_lt`
`float` comparison	`NO_UTILITY_ROM___float32_adi_lteq`
Conversion from `int` to `float`	`NO_UTILITY_ROM___int32_to_float32`
Conversion from `unsigned int` to `float`	`NO_UTILITY_ROM___unsigned_int32_to_float32`
`float` addition	`NO_UTILITY_ROM___float32_add`
`float` difference	`NO_UTILITY_ROM___float32_sub`
`float` division	`NO_UTILITY_ROM___float32_div`

Table 7-5: Utility ROM `libcc` Contents (Continued)

Compiler Support	Disable Macro
`float` product	`NO_UTILITY_ROM___float32_mul`
Conversion from `fract16` to `float`	`NO_UTILITY_ROM___fr16_to_float`
`double` comparison	`NO_UTILITY_ROM___float64_adi_gteq`
`double` comparison	`NO_UTILITY_ROM___float64_adi_gt`
Conversion from `int` to `double`	`NO_UTILITY_ROM___int32_to_float64`
Conversion from `long long` to `double`	`NO_UTILITY_ROM___int64_to_float64`
Conversion from `double` to `fract32`	`NO_UTILITY_ROM___float64_to_fr32`
Conversion from `double` to `fract32`	`NO_UTILITY_ROM___long_double_to_fr32`
Logical `long long` shift	`NO_UTILITY_ROM___lshftli`
`double` comparison	`NO_UTILITY_ROM___float64_adi_lteq`
`double` comparison	`NO_UTILITY_ROM___float64_adi_lt`
`long long` product	`NO_UTILITY_ROM___mul_64`
`long long` product	`NO_UTILITY_ROM___mulli3`
`long long` product	`NO_UTILITY_ROM___mullu3`
`double` multiplication	`NO_UTILITY_ROM___float64_mul`
`int` remainder	`NO_UTILITY_ROM___rem32`
`int` remainder	`NO_UTILITY_ROM___rem_s32`
`long long` remainder	`NO_UTILITY_ROM___moddi3`
`unsigned int` remainder	`NO_UTILITY_ROM___rem_u32`
`unsigned int` remainder	`NO_UTILITY_ROM___urem32`
`unsigned int` remainder	`NO_UTILITY_ROM___umoddi3`
Switch statement support	`NO_UTILITY_ROM___spswitch32`
Switch statement support	`NO_UTILITY_ROM___spswitch64`
`double` subtraction	`NO_UTILITY_ROM___float64_sub`
Conversion from `float` to `long long`	`NO_UTILITY_ROM___float32_to_longlong64`
Conversion from `float` to `unsigned long long`	`NO_UTILITY_ROM___float32_to_unsigned_longlong64`
Conversion from `unsigned int` to `double`	`NO_UTILITY_ROM___unsigned_int32_to_float64`
Conversion from `unsigned long long` to `double`	`NO_UTILITY_ROM___unsigned_int64_to_float64`
`fract` division	`NO_UTILITY_ROM___divrr_rnd`
`unsigned fract` division	`NO_UTILITY_ROM___divurur_rnd`

Table 7-5: Utility ROM `libcc` Contents (Continued)

Compiler Support	Disable Macro
accum multiplication support	`NO_UTILITY_ROM___muli48k`
accum multiplication support	`NO_UTILITY_ROM___mului48uk`
accum multiplication	`NO_UTILITY_ROM___mulkk`
unsigned accum multiplication	`NO_UTILITY_ROM___mulukuk`
long fract rounding	`NO_UTILITY_ROM___roundlr`
unsigned long fract rounding	`NO_UTILITY_ROM___roundulr`
accum rounding	`NO_UTILITY_ROM___roundk`
unsigned accum rounding	`NO_UTILITY_ROM___rounduk`
Conversion from float to fract16	`NO_UTILITY_ROM___float_to_fr16_rnd`
Conversion from float to accum	`NO_UTILITY_ROM___float_to_accum`
Conversion from float to accum	`NO_UTILITY_ROM___float_to_accum_rnd`
Conversion from float to unsigned fract16	`NO_UTILITY_ROM___float_to_unsigned_fr16`
Conversion from float to unsigned fract16	`NO_UTILITY_ROM___float_to_unsigned_fr16_rnd`
Conversion from float to unsigned fract32	`NO_UTILITY_ROM___float_to_unsigned_fr32`
Conversion from float to unsigned accum	`NO_UTILITY_ROM___float_to_unsigned_accum`
Conversion from float to unsigned accum	`NO_UTILITY_ROM___float_to_unsigned_accum_rnd`
Conversion from double to fract16	`NO_UTILITY_ROM___float64_to_fr16`
Conversion from double to fract16	`NO_UTILITY_ROM___float64_to_fr16_rnd`
Conversion from double to accum	`NO_UTILITY_ROM___float64_to_accum`
Conversion from double to accum	`NO_UTILITY_ROM___float64_to_accum_rnd`
Conversion from double to unsigned fract16	`NO_UTILITY_ROM___float64_to_unsigned_fr16`
Conversion from double to unsigned fract16	`NO_UTILITY_ROM___float64_to_unsigned_fr16_rnd`
Conversion from double to unsigned fract32	`NO_UTILITY_ROM___float64_to_unsigned_fr32`
Conversion from double to unsigned accum	`NO_UTILITY_ROM___float64_to_unsigned_accum`
Conversion from double to unsigned accum	`NO_UTILITY_ROM___float64_to_unsigned_accum_rnd`
long fract division	`NO_UTILITY_ROM___lrdivi`
unsigned long fract division	`NO_UTILITY_ROM___ulrdivi`
Fixed-point multiplication support	`NO_UTILITY_ROM___mul48x48_96`

Utility ROM libdsp Functions

The entire `libdsp` library is included in the ADSP-BF7xx L2 utility ROM.

Table 7-6: Functions from DSP Run-Time Library Defined in Utility ROM

Function	Header	Disable Macro
a_compress	filter.h	NO_UTILITY_ROM__a_compress
a_expand	filter.h	NO_UTILITY_ROM__a_expand
acos_fr16	math.h	NO_UTILITY_ROM__acos_fr16
acos_fr32	math.h	NO_UTILITY_ROM__acos_fr32
alog10d	math.h	NO_UTILITY_ROM__alog10d
alog10f	math.h	NO_UTILITY_ROM__alog10f
alogd	math.h	NO_UTILITY_ROM__alogd
alogf	math.h	NO_UTILITY_ROM__alogf
arg_fr16	complex.h	NO_UTILITY_ROM__arg_fr16
arg_fr32	complex.h	NO_UTILITY_ROM__arg_fr32
argd	complex.h	NO_UTILITY_ROM__argd
argf	complex.h	NO_UTILITY_ROM__argf
asin_fr16	math.h	NO_UTILITY_ROM__asin_fr16
asin_fr32	math.h	NO_UTILITY_ROM__asin_fr32
atan_fr16	math.h	NO_UTILITY_ROM__atan_fr16
atan_fr32	math.h	NO_UTILITY_ROM__atan_fr32
atan2_fr16	math.h	NO_UTILITY_ROM__atan2_fr16
atan2_fr32	math.h	NO_UTILITY_ROM__atan2_fr32
autocoh_fr16	stats.h	NO_UTILITY_ROM__autocoh_fr16
autocoh_fr32	stats.h	NO_UTILITY_ROM__autocoh_fr32
autocohd	stats.h	NO_UTILITY_ROM__autocohd
autocohf	stats.h	NO_UTILITY_ROM__autocohf
autocorr_fr16	stats.h	NO_UTILITY_ROM__autocorr_fr16
autocorr_fr32	stats.h	NO_UTILITY_ROM__autocorr_fr32
autocorrd	stats.h	NO_UTILITY_ROM__autocorrd
autocorrf	stats.h	NO_UTILITY_ROM__autocorrf
cabs_fr16	complex.h	NO_UTILITY_ROM__cabs_fr16
cabs_fr32	complex.h	NO_UTILITY_ROM__cabs_fr32
cabsd	complex.h	NO_UTILITY_ROM__cabsd

Table 7-6: Functions from DSP Run-Time Library Defined in Utility ROM (Continued)

Function	Header	Disable Macro
cabsf	complex.h	NO_UTILITY_ROM__cabsf
caddd	complex.h	NO_UTILITY_ROM__caddd
cartesian_fr16	complex.h	NO_UTILITY_ROM__cartesian_fr16
cartesian_fr32	complex.h	NO_UTILITY_ROM__cartesian_fr32
cartesiand	complex.h	NO_UTILITY_ROM__cartesiand
cartesianf	complex.h	NO_UTILITY_ROM__cartesianf
cdiv_fr16	complex.h	NO_UTILITY_ROM__cdiv_fr16
cdiv_fr32	complex.h	NO_UTILITY_ROM__cdiv_fr32
cdivf	complex.h	NO_UTILITY_ROM__cdivf
cdivd (revision 1.0 only)	complex.h	NO_UTILITY_ROM__cdivd
cexpd	complex.h	NO_UTILITY_ROM__cexpd
cexpf	complex.h	NO_UTILITY_ROM__cexpf
cfft_fr16	filter.h	NO_UTILITY_ROM__cfftN_scaling_fr16
cfft_fr32	filter.h	NO_UTILITY_ROM__cfft_fr32
cfft2d_fr16	filter.h	NO_UTILITY_ROM__cfft2d_fr16
cfft2d_fr32	filter.h	NO_UTILITY_ROM__cfft2d_fr32
cfftf_fr16	filter.h	NO_UTILITY_ROM__cfftf_fr16
cfftf_fr32	filter.h	NO_UTILITY_ROM__cfftf_fr32
cfir_fr16	filter.h	NO_UTILITY_ROM__cfir_fr16
cfir_fr32	filter.h	NO_UTILITY_ROM__cfir_fr32
clip	math.h	NO_UTILITY_ROM__clip
clip_fr16	math.h	NO_UTILITY_ROM__clip_fr16
cmatmmlt_fr16	matrix.h	NO_UTILITY_ROM__cmatmmlt_fr16
cmatmmlt_fr32	matrix.h	NO_UTILITY_ROM__cmatmmlt_fr32
cmatmmltd	matrix.h	NO_UTILITY_ROM__cmatmmltd
cmatmmltf	matrix.h	NO_UTILITY_ROM__cmatmmltf
cmltd	complex.h	NO_UTILITY_ROM__cmltd
coeff_iirdf1_fr16	filter.h	NO_UTILITY_ROM__coeff_iirdf1_fr16
coeff_iirdf1_fr32	filter.h	NO_UTILITY_ROM__coeff_iirdf1_fr32
conjd	complex.h	NO_UTILITY_ROM__conjd
conv2d_fr16	filter.h	NO_UTILITY_ROM__conv2d_fr16
conv2d_fr32	filter.h	NO_UTILITY_ROM__conv2d_fr32

Table 7-6: Functions from DSP Run-Time Library Defined in Utility ROM (Continued)

Function	Header	Disable Macro
conv2d3x3_fr16	filter.h	NO_UTILITY_ROM__conv2d3x3_fr16
conv2d3x3_fr32	filter.h	NO_UTILITY_ROM__conv2d3x3_fr32
convolve_fr16	filter.h	NO_UTILITY_ROM__convolve_fr16
convolve_fr32	filter.h	NO_UTILITY_ROM__convolve_fr32
copysign_fr16	math.h	NO_UTILITY_ROM__copysign_fr16
copysign_fr32	math.h	NO_UTILITY_ROM__copysign_fr32
copysignd	math.h	NO_UTILITY_ROM__copysignd
copysignf	math.h	NO_UTILITY_ROM__copysignf
cos_fr16	math.h	NO_UTILITY_ROM__cos_fr16
cos_fr32	math.h	NO_UTILITY_ROM__cos_fr32
cotd	math.h	NO_UTILITY_ROM__cotd
cotf	math.h	NO_UTILITY_ROM__cotf
countones	math.h	NO_UTILITY_ROM__countones
crosscoh_fr16	stats.h	NO_UTILITY_ROM__crosscoh_fr16
crosscoh_fr32	stats.h	NO_UTILITY_ROM__crosscoh_fr32
crosscohd	stats.h	NO_UTILITY_ROM__crosscohd
crosscohf	stats.h	NO_UTILITY_ROM__crosscohf
crosscorr_fr16	stats.h	NO_UTILITY_ROM__crosscorr_fr16
crosscorr_fr32	stats.h	NO_UTILITY_ROM__crosscorr_fr32
crosscorrd	stats.h	NO_UTILITY_ROM__crosscorrd
crosscorrf	stats.h	NO_UTILITY_ROM__crosscorrf
csubd	complex.h	NO_UTILITY_ROM__csubd
cvecdot_fr16	vector.h	NO_UTILITY_ROM__cvecdot_fr16
cvecdot_fr32	vector.h	NO_UTILITY_ROM__cvecdot_fr32
cvecdotd	vector.h	NO_UTILITY_ROM__cvecdotd
cvecdotf	vector.h	NO_UTILITY_ROM__cvecdotf
cvecsadd_fr16	vector.h	NO_UTILITY_ROM__cvecsadd_fr16
cvecsadd_fr32	vector.h	NO_UTILITY_ROM__cvecsadd_fr32
cvecsaddd	vector.h	NO_UTILITY_ROM__cvecsaddd
cvecsaddf	vector.h	NO_UTILITY_ROM__cvecsaddf
cvecsmlt_fr16	vector.h	NO_UTILITY_ROM__cvecsmlt_fr16
cvecsmlt_fr32	vector.h	NO_UTILITY_ROM__cvecsmlt_fr32

Table 7-6: Functions from DSP Run-Time Library Defined in Utility ROM (Continued)

Function	Header	Disable Macro
cvecsmltd	vector.h	NO_UTILITY_ROM__cvecsmltd
cvecsmltf	vector.h	NO_UTILITY_ROM__cvecsmltf
cvecssub_fr16	vector.h	NO_UTILITY_ROM__cvecssub_fr16
cvecssub_fr32	vector.h	NO_UTILITY_ROM__cvecssub_fr32
cvecssubd	vector.h	NO_UTILITY_ROM__cvecssubd
cvecssubf	vector.h	NO_UTILITY_ROM__cvecssubf
cvecvadd_fr16	vector.h	NO_UTILITY_ROM__cvecvadd_fr16
cvecvadd_fr32	vector.h	NO_UTILITY_ROM__cvecvadd_fr32
cvecvaddd	vector.h	NO_UTILITY_ROM__cvecvaddd
cvecvaddf	vector.h	NO_UTILITY_ROM__cvecvaddf
cvecvmlt_fr16	vector.h	NO_UTILITY_ROM__cvecvmlt_fr16
cvecvmlt_fr32	vector.h	NO_UTILITY_ROM__cvecvmlt_fr32
cvecvmltd	vector.h	NO_UTILITY_ROM__cvecvmltd
cvecvmltf	vector.h	NO_UTILITY_ROM__cvecvmltf
cvecvsub_fr16	vector.h	NO_UTILITY_ROM__cvecvsub_fr16
cvecvsub_fr32	vector.h	NO_UTILITY_ROM__cvecvsub_fr32
cvecvsubd	vector.h	NO_UTILITY_ROM__cvecvsubd
cvecvsubf	vector.h	NO_UTILITY_ROM__cvecvsubf
expd	math.h	NO_UTILITY_ROM__expd
expf	math.h	NO_UTILITY_ROM__expf
fclipd	math.h	NO_UTILITY_ROM__fclipd
fclipf	math.h	NO_UTILITY_ROM__fclipf
fft_magnitude_fr16	filter.h	NO_UTILITY_ROM__fft_magnitude_fr16
fft_magnitude_fr32	filter.h	NO_UTILITY_ROM__fft_magnitude_fr32
fir_decima_fr16	filter.h	NO_UTILITY_ROM__fir_decima_fr16
fir_decima_fr32	filter.h	NO_UTILITY_ROM__fir_decima_fr32
fir_fr16	filter.h	NO_UTILITY_ROM__fir_fr16
fir_fr32	filter.h	NO_UTILITY_ROM__fir_fr32
fir_interp_fr16	filter.h	NO_UTILITY_ROM__fir_interp_fr16
fir_interp_fr32	filter.h	NO_UTILITY_ROM__fir_interp_fr32
gen_bartlett_fr16	window.h	NO_UTILITY_ROM__gen_bartlett_fr16
gen_bartlett_fr32	window.h	NO_UTILITY_ROM__gen_bartlett_fr32

Table 7-6: Functions from DSP Run-Time Library Defined in Utility ROM (Continued)

Function	Header	Disable Macro
gen_blackman_fr16	window.h	NO_UTILITY_ROM__gen_blackman_fr16
gen_blackman_fr32	window.h	NO_UTILITY_ROM__gen_blackman_fr32
gen_gaussian_fr16	window.h	NO_UTILITY_ROM__gen_gaussian_fr16
gen_gaussian_fr32	window.h	NO_UTILITY_ROM__gen_gaussian_fr32
gen_hamming_fr16	window.h	NO_UTILITY_ROM__gen_hamming_fr16
gen_hamming_fr32	window.h	NO_UTILITY_ROM__gen_hamming_fr32
gen_hanning_fr16	window.h	NO_UTILITY_ROM__gen_hanning_fr16
gen_hanning_fr32	window.h	NO_UTILITY_ROM__gen_hanning_fr32
gen_harris_fr16	window.h	NO_UTILITY_ROM__gen_harris_fr16
gen_harris_fr32	window.h	NO_UTILITY_ROM__gen_harris_fr32
gen_kaiser_fr16	window.h	NO_UTILITY_ROM__gen_kaiser_fr16
gen_kaiser_fr32	window.h	NO_UTILITY_ROM__gen_kaiser_fr32
gen_rectangular_fr16	window.h	NO_UTILITY_ROM__gen_rectangular_fr16
gen_rectangular_fr32	window.h	NO_UTILITY_ROM__gen_rectangular_fr32
gen_triangle_fr16	window.h	NO_UTILITY_ROM__gen_triangle_fr16
gen_triangle_fr32	window.h	NO_UTILITY_ROM__gen_triangle_fr32
gen_vonhann_fr16	window.h	NO_UTILITY_ROM__gen_vonhann_fr16
histogram_fr16	stats.h	NO_UTILITY_ROM__histogram_fr16
histogram_fr32	stats.h	NO_UTILITY_ROM__histogram_fr32
histogramd	stats.h	NO_UTILITY_ROM__histogramd
histogramf	stats.h	NO_UTILITY_ROM__histogramf
ifft_fr16	filter.h	NO_UTILITY_ROM__ifftN_scaling_fr16
ifft_fr32	filter.h	NO_UTILITY_ROM__ifft_fr32
ifft2d_fr16	filter.h	NO_UTILITY_ROM__ifft2d_fr16
ifft2d_fr32	filter.h	NO_UTILITY_ROM__ifft2d_fr32
ifftf_fr16	filter.h	NO_UTILITY_ROM__ifftf_fr16
ifftf_fr32	filter.h	NO_UTILITY_ROM__ifftf_fr32
iir_fr16	filter.h	NO_UTILITY_ROM__iir_fr16
iir_fr32	filter.h	NO_UTILITY_ROM__iir_fr32
iirdf1_fr16	filter.h	NO_UTILITY_ROM__iirdf1_fr16
iirdf1_fr32	filter.h	NO_UTILITY_ROM__iirdf1_fr32
lcountones	math.h	NO_UTILITY_ROM__lcountones

Table 7-6: Functions from DSP Run-Time Library Defined in Utility ROM (Continued)

Function	Header	Disable Macro
llclip	math.h	NO_UTILITY_ROM__llclip
llcountones	math.h	NO_UTILITY_ROM__llcountones
matmmlt_fr16	matrix.h	NO_UTILITY_ROM__matmmlt_fr16
matmmlt_fr32	matrix.h	NO_UTILITY_ROM__matmmlt_fr32
matmmltd	matrix.h	NO_UTILITY_ROM__matmmltd
matmmltf	matrix.h	NO_UTILITY_ROM__matmmltf
mean_fr16	stats.h	NO_UTILITY_ROM__mean_fr16
mean_fr32	stats.h	NO_UTILITY_ROM__mean_fr32
meand	stats.h	NO_UTILITY_ROM__meand
meanf	stats.h	NO_UTILITY_ROM__meanf
meansf	stats.h	NO_UTILITY_ROM__meansf
mu_compress	filter.h	NO_UTILITY_ROM__mu_compress
mu_expand	filter.h	NO_UTILITY_ROM__mu_expand
normd	complex.h	NO_UTILITY_ROM__normd
normf	complex.h	NO_UTILITY_ROM__normf
polar_fr16	complex.h	NO_UTILITY_ROM__polar_fr16
polar_fr32	complex.h	NO_UTILITY_ROM__polar_fr32
polard	complex.h	NO_UTILITY_ROM__polard
polarf	complex.h	NO_UTILITY_ROM__polarf
rfft_fr16	filter.h	NO_UTILITY_ROM__rfftN_scaling_fr16
rfft_fr32	filter.h	NO_UTILITY_ROM__rfft_fr32
rfft2d_fr16	filter.h	NO_UTILITY_ROM__rfft2d_fr16
rfft2d_fr32	filter.h	NO_UTILITY_ROM__rfft2d_fr32
rfftf_fr16	filter.h	NO_UTILITY_ROM__rfftf_fr16
rfftf_fr32	filter.h	NO_UTILITY_ROM__rfftf_fr32
rms_fr16	stats.h	NO_UTILITY_ROM__rms_fr16
rms_fr32	stats.h	NO_UTILITY_ROM__rms_fr32
rmsd	stats.h	NO_UTILITY_ROM__rmsd
rmsf	stats.h	NO_UTILITY_ROM__rmsf
rsqrtd	math.h	NO_UTILITY_ROM__rsqrtd
rsqrtf	math.h	NO_UTILITY_ROM__rsqrtf
sin_fr16	math.h	NO_UTILITY_ROM__sin_fr16

Table 7-6: Functions from DSP Run-Time Library Defined in Utility ROM (Continued)

Function	Header	Disable Macro
sin_fr32	math.h	NO_UTILITY_ROM__sin_fr32
sqrt_fr16	math.h	NO_UTILITY_ROM__sqrt_fr16
sqrt_fr32	math.h	NO_UTILITY_ROM__sqrt_fr32
tan_fr16	math.h	NO_UTILITY_ROM__tan_fr16
tan_fr32	math.h	NO_UTILITY_ROM__tan_fr32
transpm_fr16	matrix.h	NO_UTILITY_ROM__transpm_fr16
transpm128	matrix.h	NO_UTILITY_ROM__transpm128
transpm16	matrix.h	NO_UTILITY_ROM__transpm16
transpm32	matrix.h	NO_UTILITY_ROM__transpm32
transpm64	matrix.h	NO_UTILITY_ROM__transpm64
transpmd	matrix.h	NO_UTILITY_ROM__transpmd
transpmf	matrix.h	NO_UTILITY_ROM__transpmf
twidfft2d_fr16	filter.h	NO_UTILITY_ROM__twidfft2d_fr16
twidfft2d_fr32	filter.h	NO_UTILITY_ROM__twidfftf_fr32
twidfftf_fr16	filter.h	NO_UTILITY_ROM__twidfftf_fr16
twidfftf_fr32	filter.h	NO_UTILITY_ROM__twidfftf_fr32
twidfftrad2_fr16	filter.h	NO_UTILITY_ROM__twidfftrad2_fr16
twidfftrad2_fr32	filter.h	NO_UTILITY_ROM__twidfftrad2_fr32
var_fr16	stats.h	NO_UTILITY_ROM__var_fr16
var_fr32	stats.h	NO_UTILITY_ROM__var_fr32
vard	stats.h	NO_UTILITY_ROM__vard
varf	stats.h	NO_UTILITY_ROM__varf
vecdot_fr16	vector.h	NO_UTILITY_ROM__vecdot_fr16
vecdot_fr32	vector.h	NO_UTILITY_ROM__vecdot_fr32
vecdotd	vector.h	NO_UTILITY_ROM__vecdotd
vecdotf	vector.h	NO_UTILITY_ROM__vecdotf
vecmax_fr16	vector.h	NO_UTILITY_ROM__vecmax_fr16
vecmax_fr32	vector.h	NO_UTILITY_ROM__vecmax_fr32
vecmaxd	vector.h	NO_UTILITY_ROM__vecmaxd
vecmaxf	vector.h	NO_UTILITY_ROM__vecmaxf
vecmaxloc_fr16	vector.h	NO_UTILITY_ROM__vecmaxloc_fr16
vecmaxloc_fr32	vector.h	NO_UTILITY_ROM__vecmaxloc_fr32

Table 7-6: Functions from DSP Run-Time Library Defined in Utility ROM (Continued)

Function	Header	Disable Macro
vecmaxlocd	vector.h	NO_UTILITY_ROM__vecmaxlocd
vecmaxlocf	vector.h	NO_UTILITY_ROM__vecmaxlocf
vecmin_fr16	vector.h	NO_UTILITY_ROM__vecmin_fr16
vecmin_fr32	vector.h	NO_UTILITY_ROM__vecmin_fr32
vecmind	vector.h	NO_UTILITY_ROM__vecmind
vecminf	vector.h	NO_UTILITY_ROM__vecminf
vecminloc_fr16	vector.h	NO_UTILITY_ROM__vecminloc_fr16
vecminloc_fr32	vector.h	NO_UTILITY_ROM__vecminloc_fr32
vecminlocd	vector.h	NO_UTILITY_ROM__vecminlocd
vecminlocf	vector.h	NO_UTILITY_ROM__vecminlocf
vecsadd_fr16	vector.h	NO_UTILITY_ROM__vecsadd_fr16
vecsadd_fr32	vector.h	NO_UTILITY_ROM__vecsadd_fr32
vecsaddd	vector.h	NO_UTILITY_ROM__vecsaddd
vecsaddf	vector.h	NO_UTILITY_ROM__vecsaddf
vecsmlt_fr16	vector.h	NO_UTILITY_ROM__vecsmlt_fr16
vecsmlt_fr32	vector.h	NO_UTILITY_ROM__vecsmlt_fr32
vecsmltd	vector.h	NO_UTILITY_ROM__vecsmltd
vecsmltf	vector.h	NO_UTILITY_ROM__vecsmltf
vecssub_fr16	vector.h	NO_UTILITY_ROM__vecssub_fr16
vecssub_fr32	vector.h	NO_UTILITY_ROM__vecssub_fr32
vecssubd	vector.h	NO_UTILITY_ROM__vecssubd
vecssubf	vector.h	NO_UTILITY_ROM__vecssubf
vecvadd_fr16	vector.h	NO_UTILITY_ROM__vecvadd_fr16
vecvadd_fr32	vector.h	NO_UTILITY_ROM__vecvadd_fr32
vecvaddd	vector.h	NO_UTILITY_ROM__vecvaddd
vecvaddf	vector.h	NO_UTILITY_ROM__vecvaddf
vecvmlt_fr16	vector.h	NO_UTILITY_ROM__vecvmlt_fr16
vecvmlt_fr32	vector.h	NO_UTILITY_ROM__vecvmlt_fr32
vecvmltd	vector.h	NO_UTILITY_ROM__vecvmltd
vecvmltf	vector.h	NO_UTILITY_ROM__vecvmltf
vecvsub_fr16	vector.h	NO_UTILITY_ROM__vecvsub_fr16
vecvsub_fr32	vector.h	NO_UTILITY_ROM__vecvsub_fr32

Table 7-6: Functions from DSP Run-Time Library Defined in Utility ROM (Continued)

Function	Header	Disable Macro
vecvsubd	vector.h	NO_UTILITY_ROM__vecvsubd
vecvsubf	vector.h	NO_UTILITY_ROM__vecvsubf
zero_cross_fr16	stats.h	NO_UTILITY_ROM__zero_cross_fr16
zero_cross_fr32	stats.h	NO_UTILITY_ROM__zero_cross_fr32
zero_crossd	stats.h	NO_UTILITY_ROM__zero_crossd
zero_crossf	stats.h	NO_UTILITY_ROM__zero_crossf

Index

Symbols

A

Symbols

A

B

Symbols

C

Symbols

C

Symbols

C

Symbols

C

Symbols

C

Symbols

C

Symbols

C

Symbols

C

Symbols

D

Symbols

D

Symbols

D

Symbols

E

Symbols

E

Symbols

E

Symbols

F

Symbols

G

Symbols

CCES 2.9.0 C/C++ Compiler and Library Manual for Blackfin Processors   45

Symbols

L

Symbols

M

Symbols

M

Symbols

N

Symbols

N

Symbols

N

Symbols

Symbols

P

Symbols

P

Symbols

P

Symbols

P

Symbols

P

Symbols

S

Symbols

S

Symbols

S

Symbols

S

Symbols

T

Symbols

T

Symbols

T

Symbols

U

Symbols

U

Symbols

V

W

wchar_t data type...4–22
weak_entry pragma...2–207
weak entry.. 2–23

Symbols

-Werror-limit (maximum compiler errors) compiler switch....
....2–55
-Werror-warnings (treat warnings as errors) compiler switch..
....2–55

W

window
    cosine.. 5–95
    functions.. 5–28
    generators.. 5–28
window.h header file..5–28
Windows shortcuts... 2–63
    expanding...2–25
    not recognizing..2–36

Symbols

-Wmis_suppress rule_number C compiler switch............2–58
-Wmis_warn rule_number C compiler switch.................2–58

W

word alignment
    data buffer... 3–17
workarounds
    anomaly management....................................2–67,2–69
    enabling..2–68
    interaction between -si-revision, -workaround and -no-
        workaround..2–69
    isr-imask-check...2–191
    list of valid workarounds...2–69
    use of the -si-revision switch.................................. 2–68
    use of the -workaround switch................................2–68
    using the -no-workaround switch........................... 2–69

Symbols

-workaround workaround_id...2–56
-workaround workaround_id compiler switch.......2–56,2–68
-Wremarks (enable diagnostic remarks) compiler switch........
....2–55,3–4

-Wremarks (enable diagnostic warnings) compiler switch.......
....2–55

W

writing
    array elements..3–30
    preprocessor macros..2–239

Symbols

-Wterse (enable terse warnings) compiler switch............. 2–56
-xref (cross-reference list) compiler switch.....................2–56

Z

zero_cross_fr16 function..5–145
zero_cross_fr32 function..5–145
zero_cross_fx16 function.. 5–145
zero_cross_fx32 function.. 5–145
zero_crossd function...5–145
zero_cross function..5–145
ZERO_INIT qualifier.................................... 2–204,2–279
ZeroData binary object...2–279
zero-length arrays..2–231

Symbols

-zero-loop-counter compiler switch...............................2–57