VisualDSP++® 5.0
C/C++ Compiler and Library Manual for Blackfin® Processors

Revision 5.4, January 2011

Part Number
82-000410-03

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

ANALOG DEVICES

Copyright Information

Disclaimer

Trademark and Service Mark Notice

CONTENTS

PREFACE

COMPILER

C/C++ RUN-TIME LIBRARY

PREFACE

Thank you for purchasing Analog Devices development software for Blackfin® embedded processors.

Purpose of This Manual

The *VisualDSP++ 5.0 C/C++ Compiler and Library Manual for Blackfin Processors* 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 Description

This manual contains:

- Chapter 1, "Compiler"
  Provides information on compiler options, language extensions, C/C++/assembly interfacing, and support for C++ templates
- Chapter 2, "Achieving Optimal Performance From C/C++ Source Code"
  Shows how to optimize compiler operation.
- Chapter 3, "C/C++ Run-Time Library"
  Shows how to use library functions and provides a complete C/C++ library function reference
- Chapter 4, "DSP Run-Time Library"
  Shows how to use DSP library functions and provides a complete DSP library function reference
- Appendix A, "Programming Dual-Core Blackfin Processors"
  Provides various approaches and programming guidance for developing systems on ADSP-BF561 Blackfin processors

What's New in This Manual

This revision (5.4) of the *VisualDSP++ 5.0 C/C++ Compiler and Library Manual for Blackfin Processors* manual documents changes/additions related to the C/C++ compiler and run-time library for VisualDSP++® 5.0 and subsequent updates (up to update 9). Changes/additions to this book from revision 5.3 include:

- Embedded C Support: The compiler supports the fixed-point types `fract` and `accum` as native types. Refer to "Using Native Fixed-Point Types" on page 1-104 for more information.

- New library support for 32-bit fractional values: Many of the 16-bit fractional library routines now have accompanying 32-bit fractional variants. Refer to the respective function description pages in Chapter 3, "C/C++ Run-Time Library" and Chapter 4, "DSP Run-Time Library" for details.

- 40-bit accumulator access: The compiler now supports access to the 40-bit accumulators, via new built-in functions. For more information, see "Full-Precision Accumulator Built-In Functions" on page 1-247.

- Improved compliance with ISO/IEC standards: The compiler has optional support for a freestanding implementation of the ISO/IEC 9899:1999 C standard ("C99"), and support for a free-standing implementation of the ISO/IEC14882:2003 C++ standard ("C++ 2003"). See "Language Standards Compliance" on page 1-140 for more information.

- Stack overflow detection: The compiler can instrument generated code to detect when the stack limit is being exceeded, reducing the effort involved in debugging such problems. For multi-threaded applications, this facility requires RTOS support. For more information see "Stack Overflow Detection" on page 2-142.

- `fract32` support: The majority of functions in Chapter 2, "Achieving Optimal Performance From C/C++ Source Code" now have variants that support the `fract32` data type.

- Corrections of typographic errors and reported document errata.

Technical or Customer Support

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

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

Supported Processors

The name "*Blackfin*" refers to a family of 16-bit, embedded processors. For a complete list of processors supported by VisualDSP++® 5.0, refer to VisualDSP++ online Help.

Product Information

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

Analog Devices Web Site

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

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

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

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

VisualDSP++ Online Documentation

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

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

Each documentation file type is described as follows.

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

Technical Library CD

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

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

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

EngineerZone

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

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

Social Networking Web Sites

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

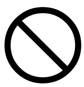
- Twitter: `http://twitter.com/blackfin`
- LinkedIn: Network with the LinkedIn group, Analog Devices Blackfin: `http://www.linkedin.com`

Notation Conventions

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

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

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of `this`.

Example	Description
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
	**Note:** For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	**Caution:** Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
	**Warning:** Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

1 COMPILER

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

> The code examples in this manual have been compiled using VisualDSP++ 5.0.1 (Update 1). The examples compiled with other versions of VisualDSP++ may result in build errors or different output although the highlighted algorithms stand and should continue to stand in future releases of VisualDSP++.

This chapter contains:

- "C/C++ Compiler Overview" on page 1-3
  provides an overview of the C/C++ compiler for Blackfin processors.
- "Compiler Command-Line Interface" on page 1-5
  describes the operation of the compiler as it processes programs, including input and output files and command-line switches.
- "Using Native Fixed-Point Types" on page 1-104
  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" on page 1-140
  describes how to enable the best possible compliance to the ISO/IEC 9899:1990 C standard, the ISO/IEC 9899:1999 C standard, or the ISO/IEC 14882:2003 C++ standard.

- "MISRA-C Compiler" on page 1-143
  describes the compiler support for MISRA-C:2004 Guidelines for the use of the C language in critical systems.

- "C/C++ Compiler Language Extensions" on page 1-156
  describes the `ccblkfn` compiler's extensions to the ANSI/ISO standard for the C and C++ languages.

- "Blackfin Processor-Specific Functionality" on page 1-357
  contains information that is specific to Blackfin processors only.

- "C/C++ Preprocessor Features" on page 1-401
  contains information on the preprocessor and ways to modify source compilation.

- "C/C++ Run-Time Model and Environment" on page 1-408
  contains reference information about implementation of C/C++ programs, data, and function calls in Blackfin processors.

- "C/C++ and Assembly Interface" on page 1-456
  describes how to call an assembly language subroutine from within a C/C++ program, and how to call a C/C++ function from within an assembly language program.

- "Compiler C++ Template Support" on page 1-466
  describes how templates are instantiated at compile time.

- "File Attributes" on page 1-471
  describes how file attributes help with the placement of run-time library functions.

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 VisualDSP++ 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 freestanding 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 available in the *docs/cpl_lib* directory underneath your VisualDSP++ installation and opening it in a Web browser.

RTTI and exceptions for C++ are supported, but disabled by default. See information on these switches: "-rtti" on page 1-90 and "-eh" on page 1-35.

For information on the C language standard, see any of the many reference texts on the C language. Analog Devices recommends the Bjarne Stroustrup text "*The C++ Programming Language*" from Addison Wesley Longman Publishing Co (ISBN: 0201889544) (1997) as a reference text for the C++ programming language.

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" on page 1-156.

You can specify compiler options from the **Compile** page of the **Project Options** dialog box of the VisualDSP++ Integrated Development and Debug Environment (IDDE). These selections control how the compiler processes your source files, letting you select features that include the language dialect, error reporting, and debugger output.

For more information on the VisualDSP++ environment, refer to VisualDSP++ online Help.

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" on page 1-6
- "C/C++ Compiler Command-Line Switches" on page 1-10
- "Environment Variables Used by the Compiler" on page 1-91
- "Additional Path Support" on page 1-92
- "Optimization Control" on page 1-95
- "Controlling Silicon Revision and Anomaly Workarounds Within the Compiler" on page 1-100

By default, the compiler runs with Analog Extensions for C code enabled. This means that the compiler processes source files written in ISO/IEC 899:1990 standard C language supplemented with Analog Devices extensions. Table 1-2 on page 1-8 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 Table 1-3 on page 1-9.) Table 1-4 on page 1-11 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 Table 1-5 on page 1-11. A summary of the C++-specific compiler switches appears in Table 1-6 on page 1-24. The summaries are followed by descriptions of each switch.

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.

Running the Compiler

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

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

Table 1-1 describes the command-line syntax.

Table 1-1. 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

A file name can include the directory, file name, and file extension. The compiler supports both Win32- 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).

> When file names or other switches for the compiler include spaces or other special characters, you must 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. Table 1-3 on page 1-9 lists the allowed extensions.

Examples

For example, the following command line runs `ccblkfn` with the following options:

```
ccblkfn -proc ADSP-BF535 -O -Wremarks -o program.dxe source.c
```

`-proc ADSP-BF535`	Specifies compiler instructions unique to the ADSP-BF535 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

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

```
ccblkfn -proc ADSP-BF535 -c++ source.cpp
```

`-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 Table 1-2 to perform a specified action.

Table 1-2. 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 VisualDSP++ IDDE.

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. Table 1-3 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. Table 1-3 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 Table 1-3 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.

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.

Table 1-3. 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)
`.ii .ti`	Template instantiation files – used internally by the compiler when instantiating templates
`.ipa`	Interprocedural analysis files – used internally by the compiler when performing interprocedural analysis.
`.pgo`	Execution profile generated by a simulation run. For more information, see "Using PGO in Function Profiling" in Chapter 2, Achieving Optimal Performance From C/C++ Source Code.

Table 1-3. Input and Output File Extensions (Cont'd)

File Extension	File Extension Description
`.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` (on page 1-72) 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
`.pch`	Precompiled header file
`.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

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" on page 1-91 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 Switches" (Table 1-4)
- "C/C++ Compiler Common Switches" (Table 1-5)
- "C Mode (MISRA) Compiler Switches" (Table 1-6 on page 1-24)
- "C++ Mode Compiler Switches" (Table 1-7 on page 1-25)

Table 1-4. C/C++ Mode Selection Switches

Switch Name	Description
`-c89` on page 1-26	Supports programs that conform to the ISO/IEC 9899:1990 standard. This is the default mode.
`-c99` on page 1-26	Supports programs that conform to a freestanding implementation of the ISO/IEC 9899:1999 standard with Analog Devices extensions.
`-c++` on page 1-26	Supports ANSI/ISO standard C++ with Analog Devices extensions

Table 1-5. C/C++ Compiler Common Switches

Switch Name	Description
`sourcefile` on page 1-27	This parameter specifies the file to be compiled
`-@` *`filename`* on page 1-27	Reads command-line input from the file
`-A` *`symbol`* `[`*`tokens`*`]` on page 1-27	Asserts the specified name as a predicate
`-add-debug-libpaths` on page 1-28	Link against debug-specific variants of system libraries, where available.
`-alttok` on page 1-28	Allows alternative keywords and sequences in sources
`-always-inline` on page 1-29	Treats `inline` keyword as a requirement rather than a suggestion.

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-annotate` on page 1-30	Enables assembly annotations
`-annotate-loop-instr` on page 1-30	Provides additional annotation information for the prolog, kernel and epilog of a loop
`-auto-attrs` on page 1-30	Directs the compiler to emit automatic attributes based on the files it compiles. Enabled by default.
`-bss` on page 1-30	Causes the compiler to put global zero-initialized data into a separate BSS-style section. Set by default.
`-build-lib` on page 1-31	Directs the librarian to build a library file
`-C` on page 1-31	Retains preprocessor comments in the output file
`-c` on page 1-31	Compiles and/or assembles only, but does not link
`-const-read-write` on page 1-31	Specifies that data accessed via a pointer to `const` data may be modified elsewhere
`-const-strings` on page 1-32	Directs the compiler to mark string literals as `const` qualified
`-cplbs` on page 1-32	Instructs the compiler to assume that CPLBs are active
`-D` *`macro`*`[=`*`definition`*`]` on page 1-32	Defines `macro`
`-dcplbs` on page 1-33	Instructs the compiler to assume that data CPLBs are active
`-debug-types` on page 1-33	Supports building a `.h` file directly and writing a complete set of debugging information for the header file
`-decls-weak` `-decls-strong` on page 1-33	Determines whether uninitialized global variables should be treated as definitions or declarations
`-double-size-32` `-double-size-64` on page 1-34	Selects 32- or 64-bit IEEE format for `double`. `-double-size-32` is the default mode

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-double-size-any` on page 1-34	Indicates that the resulting object can be linked with objects built with any `double` size
`-dry` on page 1-34	Displays, but does not perform, main driver actions (verbose dry run)
`-dryrun` on page 1-35	Displays, but does not perform, top-level driver actions (terse dry run)
`-E` on page 1-35	Preprocesses, but does not compile, the source file
`-ED` on page 1-35	Preprocesses and sends all output to a file
`-EE` on page 1-35	Preprocesses and compiles the source file
`-eh` on page 1-35	Enables exception handling
`-enum-is-int` on page 1-36	By default, an `enum` can have a type larger than `int`. This option ensures the `enum` type is `int`.
`-expand-symbolic-links` on page 1-37	Provides support for Cygwin path extensions within command-line switches and `#include` preprocessor directives
`-expand-windows-shortcuts` on page 1-37	Provides support for Windows shortcuts within command-line switches and `#include` preprocessor directives
`-extra-keywords` on page 1-37	Recognizes Blackfin processor extensions to ANSI/ISO standards for C (default mode)
`-extra-loop-loads` on page 1-37	Allows the compiler to read off the start or end of memory areas, within loops, to aid performance
`-fast-fp` on page 1-38	Links with the high-speed floating-point emulation library
`-file-attr` *name* on page 1-38	Adds the specified attribute *name*/*value* pair to the file(s) being compiled

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-fixed-point-io` on page 1-38	Links with a variant of the Analog Devices I/O library containing support for printing native fixed-point types in decimal format.
`-flags-asm` *switches* `-flags-compiler` *switches* `-flags-lib` *switches* `-flags-link` *switches* `-flags-mem` *switches* on page 1-39	Passes command-line switches through the compiler to other build tools
`-force-circbuf` on page 1-39	Treats array references of the form `array[i%n]` as circular buffer operations
`-force-link` on page 1-40	Forces stack frame creation for leaf functions. (defaults to ON with `-g` option set, enforced for the `-p` option)
`-fp-associative` on page 1-40	Treats floating-point multiplication and addition as associative operations
`-full-io` on page 1-40	Links with a third party, proprietary I/O library
`-full-version` on page 1-41	Displays the version number of the driver and processes invoked by the driver
`-fx-contract` on page 1-41	Sets the default mode of `FX_CONTRACT` to `ON`.
`-fx-rounding-mode-biased` on page 1-41	Sets the default mode of `FX_ROUNDING_MODE` to `BIASED`.
`-fx-rounding-mode-truncation` on page 1-41	Sets the default mode of `FX_ROUNDING_MODE` to `TRUNCATION`.
`-fx-rounding-mode-unbiased` on page 1-41	Sets the default mode of `FX_ROUNDING_MODE` to `UNBIASED`.
`-g` on page 1-42	Generates DWARF-2 debug information
`-glite` on page 1-42	Generates lightweight DWARF-2 debug information

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-guard-vol-loads` on page 1-43	Disables interrupts during volatile loads
`-H` on page 1-43	Outputs a list of included header files, but does not compile
`-HH` on page 1-43	Outputs a list of included header files and compiles.
`-h` `-help` on page 1-43	Outputs a list of command-line switches with brief syntax descriptions
`-I` *`directory`* on page 1-44	Appends *`directory`* to the standard search path
`-I-` on page 1-44	Specifies the point in the `include` directory list where the search for header files enclosed in angle brackets should begin
`-i` on page 1-45	Outputs only header details or makefile dependencies for `include` files specified in double quotes
`-icplbs` on page 1-45	Instructs the compiler to assume that instruction CPLBs are active
`-ieee-fp` on page 1-45	Links with the fully-compliant floating-point emulation library
`-implicit-pointers` on page 1-46	Demotes incompatible-pointer-type errors into discretionary warnings. Not valid when compiling in C++ mode.
`-include` *`filename`* on page 1-46	Includes named file prior to each source file
`-ipa` on page 1-47	Specifies that interprocedural analysis should be performed for optimization between translation units
`-jcs2l` on page 1-47	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`* on page 1-47	Appends *`directory`* to the standard library search path

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-l` *`library`* on page 1-47	Searches *`library`* for functions when linking
`-list-workarounds` on page 1-48	Lists all compiler-supported errata workarounds
`-M` on page 1-48	Generates `make` rules only, but does not compile
`-MD` on page 1-49	Generates `make` rules, compiles, and prints to a file
`-MM` on page 1-49	Generates `make` rules and compiles
`-Mo` *`filename`* on page 1-49	Writes dependency information to *`filename`*. This switch is used in conjunction with the `-ED` or `-MD` options.
`-Mt` *`filename`* on page 1-49	Makes dependencies, where the target is renamed as *`filename`*
`-map` *`filename`* on page 1-49	Directs the linker to generate a memory map of all symbols
`-mem` on page 1-50	Causes the compiler to invoke the Memory Initializer after linking the executable file
`-multicore` on page 1-50	Selects library versions suitable for use in a multi-core environment
`-multiline` on page 1-50	Enables string literals over multiple lines (default)
`-never-inline` on page 1-51	Ignores `inline` keyword on function definitions
`-no-alttok` on page 1-51	Disallows alternative keywords and sequences in sources
`-no-annotate` on page 1-51	Disables the annotation of assembly files
`-no-annotate-loop-instr` on page 1-52	Disables the production of additional loop annotation information by the compiler (default mode)

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-no-assume-vols-are-mmrs` on page 1-52	Directs the compiler not to apply workarounds for MMR-related silicon errata to arbitrary `volatile`-qualified memory accesses.
`-no-auto-attrs` on page 1-52	Directs the compiler not to emit automatic attributes based on the files it compiles
`-no-bss` on page 1-53	Causes the compiler to group global zero-initialized data into the same section as global data with non-zero initializers
`-no-builtin` on page 1-53	Disable recognition of `__builtin` functions
`-no-circbuf` on page 1-53	Disables the automatic generation of circular buffering code
`-no-const-strings` on page 1-53	Directs the compiler not to make string literals `const` qualified
`-no-defs` on page 1-54	Disables preprocessor definitions: macros, include directories, library directories, run-time headers, or keyword extensions
`-no-eh` on page 1-54	Disables exception-handling
`-no-expand-symbolic-links` on page 1-54	Disables support for Cygwin path extensions in command-line paths and preprocessor include directives
`-no-expand-windows-shortcuts` on page 1-54	Disables support for Windows shortcuts in command-line paths and preprocessor include directives
`-no-extra-keywords` on page 1-54	Does not define language extension keywords that could be valid C/C++ identifiers
`-no-force-link` on page 1-55	Does not create a new stack frame for leaf functions, if one can be omitted. Overrides the default for `-g`.
`-no-fp-associative` on page 1-55	Does not treat floating-point multiplication and addition as associative operations
`-no-full-io` on page 1-56	Links with the Analog Devices I/O library. Enabled by default

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-no-fx-contract` on page 1-56	Sets the default mode of `FX_CONTRACT` to `OFF`.
`-no-int-to-fract` on page 1-56	Prevents the compiler from turning integer into fractional arithmetic
`-no-jcs2l` on page 1-57	Prevents the linker from converting compiler-generated short jumps to long jumps using register `P1`
`-no-mem` on page 1-57	Causes the compiler to not invoke the Memory Initializer after linking. Set by default.
`-no-multiline` on page 1-57	Disables multiple line string literal support
`-no-progress-rep-timeout` on page 1-57	Prevents the compiler from issuing a diagnostic during excessively long compilations
`-no-sat-associative` on page 1-57	Saturating addition is not associative
`-no-saturation` on page 1-58	Causes the compiler not to introduce saturation semantics when optimizing expressions
`-no-std-ass` on page 1-58	Prevents the compiler from defining standard assertions
`-no-std-def` on page 1-58	Disables normal macro definitions and also Analog Devices keyword extensions that do not have leading underscores (__)
`-no-std-inc` on page 1-59	Searches only for preprocessor `include` header files in the current directory and in directories specified with the `-I` switch
`-no-std-lib` on page 1-59	When linking, searches for libraries only in directories specified with the `-L` switch
`-no-threads` on page 1-59	Specifies that no support is required for multi-threaded applications
`-no-workaround` *`workaround_id`* on page 1-59	Disables specific hardware anomaly workarounds within the compiler
`-no-zero-loop-counters` on page 1-60	Do not zero loop counters (`LC0` and `LC1`) on function exit

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description

`-O`
`-O1`
`-O0`
on page 1-60
	Enables (`-O` or `-O1`) or disables (`-O0`) code optimizations (uppercase "O" optionally followed by a zero or a one)

`-Oa` on page 1-60	Enables automatic function inlining
`-Ofp` on page 1-60	Offsets the frame pointer to allow more short load and store instructions. Reduces debugger capabilities, when used with `-g`.
`-Og` on page 1-61	Enables a compiler mode that performs optimizations while still preserving the debugging information
`-Os` on page 1-61	Optimizes the file to decrease code size
`-Ov` *`num`* on page 1-61	Controls speed versus size optimizations
`-o` *`filename`* on page 1-63	Specifies the output file name
`-overlay` on page 1-64	Disables the propagation of register information between functions and forces the compiler to assume that all functions clobber all scratch registers
`-overlay-clobbers` *`registers`* on page 1-64	Specifies the registers assumed to be clobbered by an overlay manager
`-P` on page 1-65	Preprocesses, but does not compile, the source file; output does not contain `#line` directives
`-PP` on page 1-65	Preprocesses and compiles the source file; output does not contain `#line` directives.

`-p1`
`-p2`
on page 1-65
	Generates profiling instrumentation

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-path-asm` *filename* `-path-compiler` *filename* `-path-lib` *filename* `-path-link` *filename* on page 1-65	Uses the specified directory as the location of the specified compilation tool (assembler, compiler, library builder, or linker)
`-path-install` *directory* on page 1-66	Uses the specified directory as the location of all compilation tools
`-path-output` *directory* on page 1-66	Specifies the location of non-temporary files
`-path-temp` *directory* on page 1-66	Specifies the location of temporary files
`-pch` on page 1-66	Enables automatic generation and use of precompiled header files
`-pchdir` *directory* on page 1-66	Specifies an alternative directory to `PCHRepository` in which to store precompiled header files
`-pgo-session` *session-id* on page 1-67	Used with profile-guided optimization
`-pguide` on page 1-67	Adds instrumentation for the gathering of a profile as the first stage of performing profile-guided optimization
`-pplist` *filename* on page 1-68	Outputs a raw preprocessed listing to the specified file
`-proc` *processor* on page 1-68	Specifies a processor for which the compiler should produce suitable code
`-progress-rep-func` on page 1-69	Issues a diagnostic message each time the compiler starts compiling a new function. Equivalent to `-Wwarn=cc1472`.
`-progress-rep-opt` on page 1-69	Issues a diagnostic message each time the compiler starts a new optimization pass on the current function. Equivalent to `-Wwarn=cc1473`.
`-progress-rep-timeout` on page 1-70	Issues a diagnostic message if the compiler exceeds a time limit during compilation.

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-progress-rep-timeout-secs` *`secs`* on page 1-70	Specifies how many seconds must elapse during a compilation before the compiler issues a diagnostic on the length of compilation.
`-R` *`directory`* on page 1-70	Appends *`directory`* to the standard search path for source files
`-R-` on page 1-71	Removes all directories from the source file search directory list
`-reserve` *`register(s)`* on page 1-71	Reserves certain registers from compiler use. **Note:** Reserving registers can have a detrimental effect on the compiler's optimization capabilities.
`-S` on page 1-71	Stops compilation before running the assembler
`-s` on page 1-71	When linking, removes debugging information from the output executable file
`-sat-associative` on page 1-71	Saturating addition is associative
`-save-temps` on page 1-72	Saves intermediate files
`-sdram` on page 1-72	Instructs the compiler to assume that at least bank 0 of external SDRAM will be present and enabled
`-section id=section_name` on page 1-72	Orders the compiler to place data/program of type "`id`" into the section "`section_name`"
`-show` on page 1-73	Displays the driver command-line information
`-signed-bitfield` on page 1-74	Makes the default type for `int bitfields` signed
`-signed-char` on page 1-74	Makes the default type for `char` signed
`-si-revision` *`version`* on page 1-74	Specifies a silicon revision of the specified processor. The default setting is the latest silicon revision

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-stack-detect` on page 1-74	Causes the compiler to generate additional instructions in the generated code to detect a potential stack overflow.
`-structs-do-not-overlap` on page 1-75	Specifies that `struct` copies may use "`memcpy`" semantics, rather than the usual "`memmove`" behavior
`-syntax-only` on page 1-75	Checks the source code for compiler syntax errors, but does not write any output
`-sysdefs` on page 1-76	Instructs the driver to define preprocessor macros that describe the current user and machine
`-T` *`filename`* on page 1-76	Specifies the linker description file
`-threads` on page 1-76	Enables the support for multi-threaded applications
`-time` on page 1-77	Displays the elapsed time as part of the output information on each part of the compilation process
`-U` *`macro`* on page 1-77	Undefines *`macro`*
`-unsigned-bitfield` on page 1-77	Makes the default type for plain `int` bit-fields unsigned
`-unsigned-char` on page 1-78	Makes the default type for `char` unsigned
`-v` on page 1-78	Displays version and command-line information for all compilation tools
`-verbose` on page 1-79	Displays command-line information for all compilation tools as they process each file
`-version` on page 1-79	Displays version information for all compilation tools as they process each file
`-Werror` *`number`* `-Wremark` *`number`* `-Wsuppress` *`number`* `-Wwarn` *`number`* on page 1-79	Overrides the default severity of the specified messages (errors, remarks, or warnings)

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-Werror-limit` *number* on page 1-80	Stops compiling after reaching the specified number of errors
`-Werror-warnings` on page 1-80	Directs the compiler to treat all warnings as errors
`-Wremarks` on page 1-80	Issues compiler remarks
`-Wterse` on page 1-80	Issues the briefest form of compiler warnings, errors, and remarks
`-w` on page 1-80	Disables all warnings
`-warn-protos` on page 1-81	Issues warnings about functions without prototypes
`-workaround` *workaround_id* on page 1-81	Enables code generator workaround for specific hardware errata
`-write-files` on page 1-81	Enables compiler I/O redirection
`-write-opts` on page 1-82	Passes the user options (but not input file names) via a temporary file
`-xref` *filename* on page 1-82	Outputs cross-reference information to the specified file
`-zero-loop-counters` on page 1-83	Ensure used loop counters (`LC0` and `LC1`) are zeroed on function exit

Table 1-6. C Mode (MISRA) Compiler Switches

Switch Name	Description
-misra on page 1-83	Enables checking for MISRA-C:2004 Guidelines. Allows some relaxation of interpretation. For more information, see "Rules Descriptions" on page 1-147.
-misra-linkdir directory on page 1-84	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.
-misra-no-cross-module on page 1-84	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 on page 1-84	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 on page 1-84	Enables checking for MISRA-C:2004 Guidelines. Rules relaxed by -misra option are enforced fully by this option. For more information, see "Rules Descriptions" on page 1-147.
-misra-suppress-advisory on page 1-85	Implies -misra, but suppresses the reporting of advisory rules.
-misra-testing on page 1-85	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 on page 1-85	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 on page 1-85	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 1-7. C++ Mode Compiler Switches

Switch Name	Description
`-anach` on page 1-85	Supports some language features (anachronisms) that are prohibited by the C++ standard but still in common use
`-check-init-order` on page 1-87	Adds run-time checking to the generated code highlighting potential uninitialized external objects. For development purposes only - do not use in production code.
`-extern-inline` on page 1-87	Allows standard behavior with respect to extern inline functions.
`-friend-injection` on page 1-88	Allows non-standard behavior with respect to friend declarations. When friend names are not injected, function names are visible only when using dependent lookup. This is the default mode.
`-full-dependency-inclusion` on page 1-88	Ensures re-inclusion of implicitly included files when generating dependency information
`-ignore-std` on page 1-88	Disables namespace `std` within the C++ Standard header files
`-no-anach` on page 1-89	Disallows the use of anachronisms that are prohibited by the C++ standard
`-no-extern-inline` on page 1-89	Treats extern inline functions as though they have static linkage. This is the default mode.
`-no-friend-injection` on page 1-89	Allows standard behavior. Friend function names are visible only when using argument-dependent lookup and friend class names are never visible.
`-no-implicit-inclusion` on page 1-89	Prevents implicit inclusion of source files as a method of finding definitions of template entities to be instantiated
`-no-rtti` on page 1-90	Disables run-time type information
`-no-std-templates` on page 1-90	Disables the special lookup of names used in templates
`-rtti` on page 1-90	Enables run-time type information
`-std-templates` on page 1-90	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, use the following switches: `-alttok`, `-const-read-write`, and `-no-extra-keywords`. (See Table 1-5 on page 1-11.)

-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" on page 1-140.

> The compiler does not support the `_Complex` and `_Imaginary` keywords. Complex arithmetic in C mode is enabled by including the Analog Devices-specific header file `<complex.h>`.

-c++

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

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` switch (on page 1-35) and `-rtti` switch (on page 1-90).

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. Table 1-3 on page 1-9 lists the permitted extensions and matching compiler operations.

-@

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

The `-A name (tokens)` (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 are predefined.

Table 1-8. 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
```

> 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.

-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.

Invoke this switch with the **Use Debug System Libraries** check box located in the VisualDSP++ **Project Options** dialog box (**Link : Processor** page).

-alttok

The `-alttok` (alternative tokens) switch directs the compiler to allow digraph sequences in C and C++ source files. Additionally, the switch enables the recognition of these alternative operator keywords in C++ source files (Table 1-9).

Table 1-9. Alternative Operator Keywords

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

ⓘ To use alternative tokens in C, use #include <iso646.h>.

See also “-no-alttok” on page 1-51.

-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 the -never-inline switch (on page 1-51).

Invoke this switch with the **Always** check box located in the **Inlining** area of the VisualDSP++ **Project Options** dialog box (**Compile : General** page).

-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" on page 2-96 for more details on this feature.

Invoke this switch by selecting the **Generate assembly code annotations** check box located in the VisualDSP++ **Project Options** dialog box (**Compile** page, **General** category).

See also "-no-annotate" on page 1-51.

-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" on page 2-96 for more details on this feature.

See also "-no-annotate-loop-instr" on page 1-52.

-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" on page 1-471 for more information about attributes and what automatic attributes the compiler emits. See also the `-no-auto-attrs` switch (on page 1-52) and the `-file-attr` switch (on page 1-38).

-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 the `-no-bss` switch (on page 1-53).

-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` option (on page 1-63) 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`, `-EE`, `-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.

-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.

Invoke this switch with the **Pointers to const may point to non-const data** check box located in the **Constants** area of the VisualDSP++ **Project Options** dialog box (**Compile : Language Settings** page).

-const-strings

The `-const-strings` (`const`-qualify strings) switch directs the compiler to mark string literals as `const`-qualified. See also the `-no-const-strings` switch (on page 1-53).

Invoke this switch with the **Literal strings are const** check box located in the **Language Settings : Constants** area of the **Project Options** dialog box (**Compile : Language Settings** page).

-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 the "-icplbs" on page 1-45 switch or the "-dcplbs" on page 1-33 switch, respectively

Invoke this switch with the **CPLBs are enabled** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Processor (2)** page).

-D

The `-D` *`macro[=definition]`* (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.

Invoke this switch by using the **Preprocessor definitions** field located in the VisualDSP++ **Project Options** dialog box (**Compile : Preprocessor** page).

-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 the "-cplbs" on page 1-32 switch.

-debug-types *<file.h>*

The `-debug-types` switch builds a `.h` file directly and writes a complete set of debugging information for the header file. The `-g` option (on page 1-42) need not be specified with the `-debug-types` option because it is implied.

For example,

```
ccblkfn -debug-types anyHeader.h
```

Until the introduction of `-debug-types`, the compiler would not accept a `*.h` file as a valid input file. The implicit `-g` option writes debugging information for only those `typedefs` that are referenced in the program. The `-debug-types` option provides complete debugging information for all `typedefs` and `structs`.

-decls-{weak|strong}

The `-decls-weak` and `-decls-strong` switches control how the compiler interprets uninitialized global variable definitions, such as `int x;`.

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`.

Invoke this switch by means of the **Treat uninitialized global vars as...** check boxes located in the VisualDSP++ **Project Options** dialog box (**Compile : Processor (1)** page).

-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 "Using Data Storage Formats" on page 1-443 for more information on data types.

Invoke this switch with the **Double Size** option buttons located in the **Project Options** dialog box (**Compile : Processor (1)** page).

-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 "Using Data Storage Formats" on page 1-443 for more information on data types.

Invoke this switch with the **Allow mixing of sizes** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Processor (1)** page).

-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` switch (on page 1-63).

-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.

Invoke this switch with the **Generate preprocessed file** check box located in the **Project Options** dialog box (**Compile : General** page).

-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 (on page 1-26), 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 "Exceptions Tables Pragma" on page 1-347.

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.

Invoke this switch with the **C++ exceptions and RTTI** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Language Settings** page).

See also "-no-eh" on page 1-54.

-enum-is-int

The `-enum-is-int` switch ensures that the type of an `enum` is `int`. 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. This switch prevents the compiler from selecting a type wider than `int`.

Invoke this switch with the **Enumerated types are always int** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Language Settings** page).

-expand-symbolic-links

The `-expand-symbolic-links` (expand symbolic links) switch directs the compiler to recognize Cygwin path extensions (see "Cygwin Path Support" on page 1-93) within command-line switches and `#include` preprocessor directives. This option is disabled by default. See also the `-no-expand-symbolic-links` switch (on page 1-54).

-expand-windows-shortcuts

The `-expand-windows-shortcuts` (expand Windows shortcuts) switch directs the compiler to recognize Windows shortcuts ("Windows Shortcut Support" on page 1-92) within command-line switches and `#include` preprocessor directives. This option is disabled by default. See also the `-no-expand-windows-shortcuts` switch (on page 1-54).

-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.

Use the `-no-extra-keywords` switch (on page 1-54) to disallow support for the additional keywords. Table 1-21 on page 1-158 provides a list and a brief description of keyword extensions.

-extra-loop-loads

The `-extra-loop-loads` (improve code for loops) switch provides the compiler with extra freedom to read more memory locations than required, within a loop, in order to generate the best code. For example, if a loop indicated that the compiler should read elements `arr[0]..arr[59]` and sum them, the `-extra-loop-loads` switch would indicate that the compiler is also allowed to read element `arr[60]`.

-fast-fp

The `-fast-fp` (fast floating point) switch directs the compiler to link with the high-speed floating-point emulation library. This library relaxes some of the IEEE floating-point standard's rules for checking inputs against not-a-number (NaN) and denormalized numbers to improve performance. This switch is enabled by default. See also the `-ieee-fp` switch (on page 1-45). Refer to "Using Data Storage Formats" on page 1-443 for more information on data types.

Invoke this switch with the **High performance** option button located in the **Floating Point** area of the VisualDSP++ **Project Options** dialog box (**Link : Processor** page).

-file-attr

The `-file-attr` *`name[=value]`* (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" will be used. See "File Attributes" on page 1-471 for more information about attributes, and what automatic attributes the compiler emits. See also the `-auto-attrs` switch (on page 1-30) and the `-no-auto-attrs` switch (on page 1-52).

Invoke this switch with the **Additional attributes** text field located in the **Project Options** dialog box (**Compile : General** page).

-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 contains a faster implementation of C standard I/O than the alternative third-party I/O library (see "-full-io" on page 1-40.) but that the functionality provided is not as comprehensive. For details, refer to "stdio.h" on page 3-31.

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

Invoke this switch using the **High performance I/O with support for fixed-point types** option button located in the **I/O Libraries** area of the VisualDSP++ **Project Options** dialog box (**Link : Processor** page).

See also "-full-io" on page 1-40 and "-no-full-io" on page 1-56.

-flags{-asm | -compiler | -lib | -link | -mem} 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 Table 1-10.

Table 1-10. Switches Passed to Other Build Tools

Option	Tool
`-flags-asm`	Assembler
`-flags-compiler`	Compiler executable
`-flags-lib`	Library Builder (`elfar.exe`)
`-flags-link`	Linker
`-flags-mem`	Memory Initializer

-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" on page 1-256.

Invoke this switch with the **Even when pointer may be outside buffer range** option button located in the VisualDSP++ **Project Options** dialog box (**Compile : Language Settings** page).

-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 (on page 1-42) is selected as it improves the quality of debugging information, but can be switched off with `-no-force-link`. When `-p` (on page 1-65) is selected, this switch is always in force. See also `-no-force-link` switch (on page 1-55).

-fp-associative

The `-fp-associative` switch directs the compiler to treat floating-point multiplication and addition as associative operations. This switch is on by default.

See also "-no-fp-associative" on page 1-55.

-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" on page 3-31.

Invoke this switch using two options: the **Full I/O** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Processor (1)** page) and the **Full ANSI C Compliance** option button located in the **I/O Libraries** area of the VisualDSP++ **Project Options** dialog box (**Link : Processor** page).

See also "-no-full-io" on page 1-56.

-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" on page 1-115 for more information.

See also "-no-fx-contract" on page 1-56.

-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" on page 1-128 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" on page 1-242.

-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" on page 1-128 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" on page 1-128 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" on page 1-242.

-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.

Invoke this switch by selecting the **Generate debug information** check box in the VisualDSP++ **Project Options** dialog box (**Compile : General** page).

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. See also the `-Og` switch (on page 1-61).

-glite

The `-glite` (lightweight debugging) switch can be used on its own, or in conjunction with any of the `-g`, `-Og`, or `-debug-types` compiler switches. 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.

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.

-guard-vol-loads

The `-guard-vol-loads` (guard volatile loads) switch disables interrupts during volatile loads. A load can be interrupted before completion and restarted once the interrupt completes. If the load is to a device register, this can have undesirable side effects. The `-guard-vol-loads` switch disables interrupts before issuing a volatile load and re-enables interrupts after the load to avoid this problem.

Invoke this switch with the **Disable interrupts during volatile memory accesses** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Processor (1)** page).

-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` switch (on page 1-63) 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) switches directs the compiler to output a list of command-line switches with a brief syntax description.

-I

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

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`

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 VisualDSP++ **Project Options** dialog box (**Compile : 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.

> 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" on page 1-32 switch.

-ieee-fp

The `-ieee-fp` (slower floating point) switch directs the compiler to link with the fully-compliant floating-point emulation library. This library obeys all of the IEEE floating-point standard's rules, and incurs a performance penalty when compared with the default floating-point emulation library. See also the `-fast-fp` switch (on page 1-38). Refer to "Using Data Storage Formats" on page 1-443 for more information on data types.

Invoke this switch with the **Strict IEEE compliance** option button located in the **Floating Point** area of the VisualDSP++ **Project Options** dialog box (**Link : Processor** page).

-implicit-pointers

The `-implicit-pointers` (implicit pointer conversion) switch allows a pointer to one type to be converted to a pointer to another type without using an explicit cast. The compiler produces a discretionary warning rather than an error in such circumstances. This option is not valid when compiling in C++ mode.

For example, the following code will not compile without this switch:

```
int *foo(int *a) {
    return a;
}
int main(void) {
    char *p = 0, *r;
    r = foo(p);          /* Bad: normally produces an error */
    return 0;
}
```

In this example, both the argument to `foo` and the assignment to `r` will be faulted by the compiler. Using the `-implicit-pointers` switch converts these errors into warnings.

Invoke this switch with the **Allow incompatible pointer types** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Language Settings** page).

-include

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" on page 1-98. Specifying `-ipa` also implies setting the `-O` switch (on page 1-60).

Invoke this switch by selecting the **Interprocedural optimization** check box in the VisualDSP++ **Project Options** dialog box (**Compile : General** page).

-jcs2l

The `-jcs2l` switch requests the linker to convert compiler-generated `short` jumps to `long` jumps when necessary, but uses the `P1` register for indirect jumps/calls when `long` jumps/calls are insufficient. This switch is enabled by default.

See also "-no-jcs2l" on page 1-57.

-L

The `-L` *`directory[{,|;} directory…]`* (library search directory) switch directs the linker to append the directory (or directories) to the search path for library files.

-l

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 `.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.

-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" on page 1-100 for details of valid workarounds and the interaction of the `-si-revision` (on page 1-74), `-workaround` (on page 1-81), and `-no-workaround` (on page 1-59) 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 utility, 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 the `-Mo` switch (on page 1-49).

-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

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

-Mt

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

The `-map` *`filename`* (generate a memory map) switch directs the compiler 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-mem` switch (on page 1-39).

See also "-no-mem" on page 1-57.

-multicore

The `-multicore` switch indicates to the compiler that the application is being built for use in a dual-core environment, such as the ADSP-BF561 Blackfin processor. It indicates that both cores are operating at once, and therefore the application is linked against versions of the libraries that include locking and per-core private storage. The `-multicore` switch defines the `__ADI_MULTICORE` macro to the value "1" at both compile-time and link-time.

The `-multicore` switch is not supported in conjunction with the `-p`, `-p1`, or `-p2` switches.

Invoke this switch with the:

- **Will be linked with re-entrant libraries** check box located in the **Project Options** dialog box (**Compile : Processor** (2) page)
- **Use re-entrant multicore libraries** check box located in the **Libraries** area of the VisualDSP++ **Project Options** dialog box (**Link : Processor** page).

-multiline

The `-multiline` switch enables a compiler GNU compatibility mode, which allows string literals to span multiple lines without the need for a backslash character "`\`" at the end of each line. This is the default mode.

Invoke this switch with the **Allow multi-line character strings** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Language Settings** page).

See also "-no-multiline" on page 1-57.

-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. See also "-always-inline" on page 1-29.

Invoke this switch with the **Never** option button in the **Inlining** area of the VisualDSP++ **Project Options** dialog box (**Compile : General** page).

-no-alttok

The `-no-alttok` (disable alternative tokens) switch directs the compiler not to accept alternative operator keywords and digraph sequences in the source files. This is the default mode. For more information, see "-alttok" on page 1-28.

-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" on page 2-96 for more details on this feature.

Invoke this switch by clearing the **Generate assembly code annotations** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : General** page).

See also "-annotate" on page 1-30.

-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" on page 1-30.

-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 functions (on page 1-275) will still receive such workarounds. For more information, see "Controlling Silicon Revision and Anomaly Workarounds Within the Compiler" on page 1-100.

-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" on page 1-471 for more information about attributes, and what automatic attributes the compiler emits. See also the `-auto-attrs` switch (on page 1-30) and the `-file-attr` switch (on page 1-38).

Invoke this switch by clearing the **Auto-generated attributes** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : General** page).

-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 the `-bss` switch (on page 1-30).

-no-builtin

The `-no-builtin` (no built-in functions) switch directs the compiler not to generate short names for the built-in functions (for example, `abs()`), and to accept only the full name (for example, `__builtin_abs()`). Note that this switch influences many functions. This switch also predefines the `__NO_BUILTIN` preprocessor macro. For more information, see "Compiler Built-In Functions" on page 1-195.

Invoke this switch by selecting the **Disable built-in functions** check box in the VisualDSP++ **Project Options** dialog box (**Compile : Language Settings** page).

-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.

Invoke this switch with the **Never** option button located in the **Circular Buffer Generation** area of the VisualDSP++ **Project Options** dialog box (**Compile : Language Settings** page).

-no-const-strings

The `-no-const-strings` switch directs the compiler not to make string literals `const` qualified. This is the default. See also the `-const-strings` switch (on page 1-32).

-no-defs

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

-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 the `-eh` switch (on page 1-35) for more information.

-no-expand-symbolic-links

The `-no-expand-symbolic-links` switch directs the compiler not to recognize Cygwin path entities (see "Cygwin Path Support" on page 1-93) within command-line paths and preprocessor `#include` directives. This option is enabled by default. See also the `-expand-symbolic-links` switch (on page 1-37).

-no-expand-windows-shortcuts

The `-no-expand-windows-shortcuts` switch directs the compiler not to recognize Windows shortcut entities (see "Windows Shortcut Support" on page 1-92) within command-line paths and preprocessor `#include` directives. This option is enabled by default. See also the `-expand-windows-shortcuts` switch (on page 1-37).

-no-extra-keywords

The `-no-extra-keywords` (disable short-form keywords) switch directs the compiler not to recognize Analog Devices keyword extensions that might affect conformance to ANSI/ISO standards for the C and C++ languages. Keywords, such as `inline`, may be used as identifiers in conforming programs. Alternate keywords (prefixed with two leading underscores, such as `__inline`) continue to work.

Invoke this switch with the **Disable Analog Devices extension keywords** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Language Settings** page).

See also "-extra-keywords" on page 1-37.

-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 (on page 1-42) 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" on page 1-40.

-no-fp-associative

The `-no-fp-associative` switch directs the compiler NOT to treat floating-point multiplication and addition as associative operations.

Invoke this switch with the **Do not treat floating point operations as associative** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Language Settings** page).

See also "-fp-associative" on page 1-40.

-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" on page 1-40.) 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" on page 3-31.

This switch passes the `_ADI_LIBIO` macro to the compiler and linker. 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" on page 1-115 for more information.

See also "-fx-contract" on page 1-41.

-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 resulting fractional arithmetic expression overflows, the results may differ. Specifying the `-no-int-to-fract` switch disables this optimization and may be used to ensure compliance with the C standard where such saturation is a concern.

-no-jcs2l

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

See also "-jcs2l" on page 1-47.

-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" on page 1-50.

-no-multiline

The `-no-multiline` switch disables a compiler GNU compatibility mode, which allows string literals to span multiple lines without requiring a "`\`" at the end of each line.

Invoke this switch by clearing the **Allow multi-line character strings** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Language Settings** page).

See also "-multiline" on page 1-50.

-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 the `-progress-rep-timeout` switch (on page 1-70) and the `-progress-rep-timeout-secs` switch (on page 1-70).

-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" on page 1-71.

-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).

Invoke this switch with the **Do not introduce saturation to integer arithmetic** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Processor** (2) page).

-no-std-ass

The `-no-std-ass` (disable standard assertions) switch prevents the compiler from defining the standard assertions. See the `-A` switch (on page 1-27) 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.

Invoke this switch by selecting the **Ignore standard include paths** check box in the VisualDSP++ **Project Options** dialog box (**Compile : Preprocessor** page).

-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` switch (on page 1-47). 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 de 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. See also the `-threads` switch (on page 1-76).

-no-workaround

The `-no-workaround` *`workaround_id[,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” on page 1-100 for details of valid workarounds and the interactions of the `-si-revision`, `-workaround`, and `-no-workaround` switches.

See also “-workaround” on page 1-81.

-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 (see "-zero-loop-counters" on page 1-83) 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.

Invoke this switch by selecting the **Enable optimization** check box in the **Project Options** dialog box (**Compile : General** page).

-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` (optimize for speed versus size) switch (on page 1-61). 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`.

Invoke this switch with the **Automatic** option button located in the **Inlining** area of the VisualDSP++ **Project Options** dialog box (**Compile : General** page).

-Ofp

The `-Ofp` (frame pointer optimization) switch directs the compiler to offset the frame pointer within a function. This allows the compiler to use more short load and store instructions. Specifying `-Ofp` also implies the use of `-O`.

Specifying this switch reduces the capabilities of the debugger for source-level debugging actions when used with `-g`, since the active call frames cannot be followed beyond an active function with a frame pointer offset. The debugger facilities that are affected by the `-Ofp` switch include: call stack, step over, and step out of.

> When C++ exceptions support is enabled (by using the `-eh` switch (on page 1-35)), the `-Ofp` switch is overridden. This is necessary to allow the exceptions handling support routines to unwind the stack from the current stack frame.

Invoke this switch with the **Frame pointer optimization** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Processor (1)** page).

-Og

The `-Og` switch enables a compiler mode that attempts to perform optimizations while still preserving debugging information. It is meant as an alternative for users who want a debuggable program but are also concerned about the performance of their debuggable code.

-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

The `-Ov` *`num`* (optimize for speed versus size) switch informs the compiler of the relative importance of speed versus size, when considering whether such tradeoffs 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. The `-Ov0` optimizes all blocks for space, and `-Ov100` optimizes all blocks for speed. 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). Figure 1-1 demonstrates this relationship.

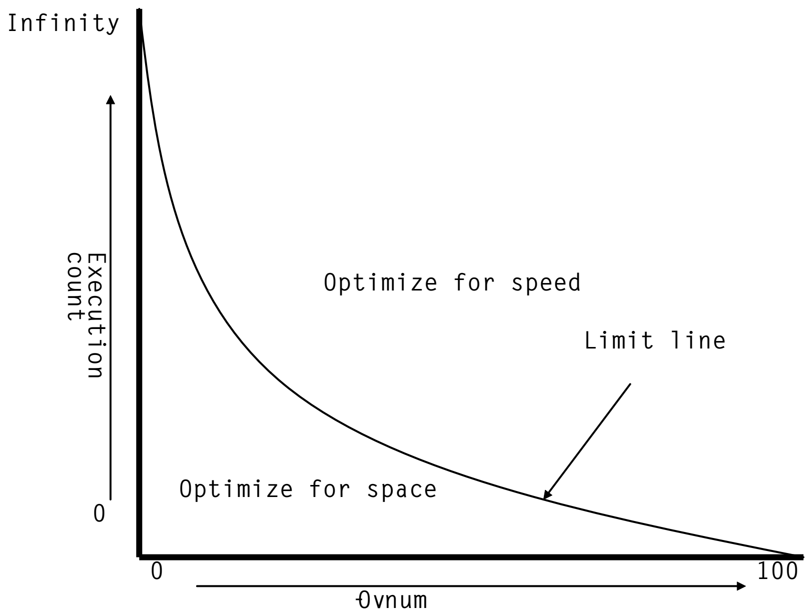


Figure 1-1. -Ov Switch Optimization Curve

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 Figure 1-1 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.

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" on page 1-95), the compiler makes estimates of the relative execution counts using heuristics.

Invoke this switch with the **Optimize for code size/speed** slider located in the VisualDSP++ **Project Options** dialog box (**Compile : General** page).

For more information, see "Using PGO in Function Profiling" in Chapter 2, Achieving Optimal Performance From C/C++ Source Code.

-o

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" on page 1-329.

-overlay-clobbers

The `-overlay-clobbers` *`clobbered-regs`* (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 (on page 1-64) is specified, or if the function is marked with `#pragma overlay` (on page 1-329).

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.

> 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[1|2]

The `-p`, `-p1`, and `-p2` (generate profiling implementation) switches direct the compiler to generate the additional instructions needed to profile the program by recording the number of cycles spent in each function.

The `-p1` switch causes the program being profiled to write the information to a file called `mon.out`. The `-p2` switch changes this behavior to write the information to the standard output file stream. The `-p` switch writes the data to `mon.out` and the standard output stream. For more information on profiling, see "Profiling With Instrumented Code" on page 1-359.

-path {-asm | -compiler | -lib | -link}

The `-path-{asm|compiler|lib|link}`*`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, librarian, and linker. 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` switch (on page 1-66).

-path-install

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.

> You can selectively override this switch with the `-path-`*`{asm|compiler|lib|link}`* switch.

-path-output

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

-path-temp

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

-pch

The `-pch` (precompiled header) switch directs the compiler to automatically generate and use precompiled header files. A precompiled output header has a `.pch` extension attached to the source file name. By default, all precompiled headers are stored in a directory called `PCHRepository`.

> Precompiled header files tend to occupy more disk space.

-pchdir

The `-pchdir` *`directory`* (locate precompiled header repository) switch specifies the location of an alternative directory for storing and invocation of precompiled header files. If the directory does not exist, the compiler creates it. Note that the `-o` (output) switch does not influence the `-pchdir` option.

-pgo-session

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 (on page 1-67), 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.

Invoke this switch with the **PGO session name** text field located in the VisualDSP++ **Project Options** dialog box (**Compile : Profile-guided Optimization** page).

For more information, see "Using PGO in Function Profiling" in Chapter 2, Achieving Optimal Performance From C/C++ Source Code.

-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.

Invoke this switch with the **Prepare application to create new profile** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Profile-guided Optimization** page).

For more information, see "Using PGO in Function Profiling" in Chapter 2, Achieving Optimal Performance From C/C++ Source Code.

-pplist

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 Table 1-11.

Table 1-11. Key Characters

Character	Meaning
`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

The `-proc` *`processor`* (target processor) switch directs the compiler to produce code suitable for the specified processor.

For example,

```
ccblkfn -proc ADSP-BF535 -o bin/p1.doj p1.asm
```

> 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 and the `__ADSPBLACKFIN__` preprocessor macro are defined as “1”. When the target is an ADSP-BF522, ADSP-BF523, ADSP-BF524, ADSP-BF525, ADSP-BF526, ADSP-BF527, ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF534, ADSP-BF536, ADSP-BF537, ADSP-BF538, ADSP-BF539, ADSP-BF542, ADSP-BF544, ADSP-BF548, ADSP-BF549, or ADSP-BF561 processor, the compiler additionally defines macro `__ADSPLPBLACKFIN__` as “1”.

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

> See also “-si-revision” on page 1-74 for more information on the silicon revision of the specified processor.

-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" on page 1-57.

-progress-rep-timeout-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" on page 1-57.

-R

The `-R` *`directory[ ,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.

ⓘ 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.

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

-reserve

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

-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.

Executable files produced by this switch are not suitable for use with the Memory Initializer (see "-mem" on page 1-50 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" on page 1-57.

-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` specified output directory or the build directory (when the `-path-output` switch (on page 1-72) is not used). See Table 1-3 on page 1-9 for a list of intermediate files.

-sdram

The `-sdram` (SDRAM is active) switch instructs the compiler to assume that at least Bank 0 of external SDRAM (the lower 32 Mbytes 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" on page 1-100.

Invoke this switch with the **SDRAM Bank 0 is in use** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Processor** (2) page).

-section

The `-section` *`id=section_name[,id=section_name...]`* 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" on page 1-193 for more details.

`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" on page 1-419.
`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 *VisualDSP++ 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 bit-fields (which have not been declared with an explicit signed or unsigned keyword) signed. This switch does not affect plain one-bit bit-fields, which are always unsigned. This is the default mode. See also the `-unsigned-bitfield` switch (on page 1-77).

-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 (on page 1-78).

-si-revision

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" on page 1-100.

-stack-detect

The `-stack-detect` (detect stack overflow) switch directs the compiler to generate the additional instructions needed to determine if the system stack has overflowed. See "Stack Overflow Detection" on page 2-142.

-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`.

Invoke this switch from the **Structs/classes do not overlap** check box in the VisualDSP++ **Project Options** dialog box (**Compile : Language Settings** page).

-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 and are used in functions with null-terminated string arguments.

The macros are defined if the system returns information for them (Table 1-12).

Table 1-12. 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

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.

When applications are built within VisualDSP++, this switch is added automatically to projects that have VDK support selected.

> The use of thread-safe libraries is necessary in conjunction with the `-threads` flag when using the VisualDSP++ kernel (VDK). The thread-safe libraries can be used with other RTOSs, but this requires the definition of various VDK interfaces.
>
> The `-threads` switch does not imply that the compiler will produce thread-safe code when compiling C/C++ source. It is the user's responsibility to employ multi-threaded programming practices in code (such as semaphores to access shared data).

See also "-no-threads" on page 1-59.

-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

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.

Invoke this switch by entering macro names to be undefined, separated by commas, in the **Preprocessor undefines** field in the **Project Options** dialog box (**Compile : Preprocessor** page).

-unsigned-bitfield

The `-unsigned-bitfield` (make plain bit-fields unsigned) switch directs the compiler to make bit-fields (which have not been declared with an explicit signed or unsigned keyword) unsigned. This switch does not affect plain one-bit bit-fields, which are always unsigned.

For example, given the declaration

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

Table 1-13 lists the `bitfield` values.

Table 1-13. Bit-field Values

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

See also the `-signed-bitfields` switch (on page 1-74).

-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.

-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{error|remark|suppress|warn}

The `-Werror`, `-Wremark`, `-Wsuppress`, and `-Wwarn` *`number[, number...]`* (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.

> 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 window and in Help, error codes sometimes begin with a leading zero (for example, cc0025). If you try to suppress error codes with `-W{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.

-Werror-limit

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.

Invoke this switch by selecting the **Enable remarks** check box in the VisualDSP++ **Project Options** dialog box (**Compile : Warning** page).

-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.

> 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 will be suppressed; otherwise, it will not be suppressed.

Invoke this switch by selecting the **Disable all warnings and remarks** check box in the VisualDSP++ **Project Options** dialog box (**Compile : Warning** page).

-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.

Invoke this switch with the **Function declarations without prototypes** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Warning** page).

-workaround

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" on page 1-100 for details of valid workarounds and the interaction of the `-si-revision`, `-workaround`, and `-no-workaround` switches.

See also "-no-workaround" on page 1-59.

-write-files

The `-write-files` (enable driver I/O redirection) switch directs the compiler driver to redirect the file name portions of its command line through a temporary file. This technique helps to handle long file names, which can make the compiler driver's command line too long for some operating systems.

ⓘ This switch is deprecated.

-write-opts

The `-write-opts` (user options) switch directs the compiler to pass the user options (but not the input file names) to the main driver via a temporary file which can help if the resulting main driver command line is too long.

This switch is deprecated.

-xref

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.

```
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 Table 1-14.

Table 1-14. ref-code Characters

Character	Meaning
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)

> ⓘ The compiler's `-xref` switch differs from the linker's `-xref` switch. Refer to the *VisualDSP++ 5.0 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 the "Hardware Loops" section of the *Blackfin Processor Programming Reference*). 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" on page 1-60.

C Mode (MISRA) Compiler Switch Descriptions

The following MISRA switches apply only to the C compiler.
See "MISRA-C Compiler" on page 1-143 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 "Rules Descriptions" on page 1-147.

The `-misra` switch is not supported in conjunction with the `-w` and `-Werror|suppress|warn` switches. The switch predefines the `_MISRA_RULES` preprocessor macro.

-misra-linkdir

The `-misra-linkdir` *`directory`* 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 the `-w` and `-Werror|remark|suppress|warn` switches.

-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 the `-w` and `-Werror|remark|suppress|warn` switches.

-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 “Rules Descriptions” on page 1-147 for more detail.

The `-misra-strict` switch is not supported in conjunction with the `-w` and `-Werror|remark|suppress|warn` switches. 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 the `-w` and `-Werror|remark|suppress|warn` switches.

-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 the `-w` and `-Werror|remark|suppress|warn` switches.

-Wmis_suppress

The `-Wmis_suppress` *`rule_number [, rule_number ]`* 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

The `-Wmis_warn` *`rule_number [, rule_number ]`* 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.

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. This is the default mode. Use the `-no-anach` switch for greater standard compliance.

The following anachronisms are accepted in the default C++ mode:

- 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++()` and `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" on page 1-89.

-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 will output 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 will generate output to `stderr` to indicate that uses of such objects are unsafe.

ⓘ This switch generates extra code to aid development. Do not use this switch when building production systems.

Invoke this switch with the **Check initialization order** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Language Settings** page).

-extern-inline

The `-extern-inline` switch directs the compiler to conform to the ISO/IEC 14882:2003 standard with respect to inline functions that are non-static. If the definition of the functions need to be retained, then the compiler will ensure that there is a unique entry point.

See also "-no-extern-inline" on page 1-89.

-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 will be injected into the scope enclosing the class containing the friend declaration.

See also "-no-friend-injection" on page 1-89.

-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++ source files are compiled more than once with different macro guards.

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

-ignore-std

The `-ignore-std` switch provides backwards compatibility to earlier versions of VisualDSP C++, which did not use namespace `std` to guard and encode C++ Standard library names. By default, the header files and libraries now use namespace `std`.

Invoke this switch by clearing the **Use std:: namespace** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Language Settings** page).

-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. See the `-anach` switch (on page 1-85) for a full description of these features.

-no-extern-inline

The `-no-extern-inline` switch directs the compiler to treat all inline functions as `static`. If the function definition needs to be retained, an external entry point is not generated. This is the default mode.

See also "-extern-inline" on page 1-87.

-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" on page 1-88.

-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.

-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" on page 1-90.

-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). This is the default.

See also "-std-templates" on page 1-90.

-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.

Invoke this switch with the **C++ exceptions and RTTI** check box located in the VisualDSP++ **Project Options** dialog box (**Compile : Language Settings** page).

See also "-no-rtti" on page 1-90.

-std-templates

The `-std-templates` switch enables dependent name processing, that is, the special lookup of names used in templates as required by the C++ standard.

See also "-no-std-templates" on page 1-90.

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.

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 executable files 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 VisualDSP++ 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.

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 (on page 1-37), and disable it with the `-no-expand-windows-shortcuts` switch (on page 1-54). 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-BF533 c:\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.

ⓘ The Cygwin environment is not part of VisualDSP++. It is provided by Red Hat, Inc. and can be downloaded from their Web site.

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 the 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
```

The compiler supports local symbolic links only. VisualDSP++ 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 file `c:\src\blackfin\test.c` 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` will then be 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
```

The compiler supports local Cygwin mounts only. VisualDSP++ does not support Cygwin mounts to remote devices and machines.

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 VisualDSP++ project options) or pragmas.

Refer to "Achieving Optimal Performance From C/C++ Source Code" on page 2-1 for information on how to obtain maximal code performance from the compiler.

The compiler's optimization capabilities are described in "Optimization Levels" on page 1-95 and "Interprocedural Analysis" on page 1-98.

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. See "-g" on page 1-42 and "-Og" on page 1-61 for more information.

- **Default**
  The compiler does not perform any optimization by default when none of the compiler optimization switches are used (or enabled in the VisualDSP++ **Project Options** dialog box). Default optimization level can be enabled using the `optimize_off` pragma (on page 1-297).

- **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]" on page 1-60, "-Os" on page 1-61, "-Ov" on page 1-61, and "-Og" on page 1-61.

  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" on page 1-297. The `-Ofp` compiler switch directs the compiler to offset the frame pointer if doing so allows more 16-bit instructions to be used. Offsetting the frame pointer means the function does not conform to the Application Binary Interface (ABI), but allows the compiler to produce smaller code, which, in turn, allows for more multi-issue instructions. Since the ABI is affected, the debugger would be unable to interpret the resulting frame structure. See "-Ofp" on page 1-60 for more information.

- **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" on page 1-67 for more information. Note that PGO is supported in the simulator only.

  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.

  You can create, edit, and delete data sets through the VisualDSP++ IDDE and then "run" the data sets with the click of one button to produce an optimized application. The PGO operation is handled via a the **Manage Data Sets** dialog box in the VisualDSP++ IDDE via: **Tools -> PGO -> Manage Data Sets**.

  For more information, see "Using Profile-Guided Optimization" on page 2-9.

  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" on page 1-358.

- **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” on page 1-60, “-Ov” on page 1-61, “-O[0|1]” on page 1-60, and “Function Inlining” on page 1-159 for more information.

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” on page 1-47, “-O[0|1]” on page 1-60, and “Interprocedural Analysis” on page 1-98 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” on page 1-287.)

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 “Achieving Optimal Performance From C/C++ Source Code” on page 2-1.

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 VisualDSP++ **Project Options** dialog box, or by specifying the `-ipa` command-line switch (on page 1-47).

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 reinvokes the compiler to perform the new optimizations, recompiling source files where necessary, to make use of gathered information.

> 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` switch (on page 1-74), which automatically controls the use of compiler workarounds.

> The compiler cannot apply errata workarounds to code inside `asm()` constructs.

When developing using the VisualDSP++ IDDE, the silicon revision within a project is set to a default value of `Automatic`. Using a silicon revision of `Automatic` will select a parameter value for the `-si-revision` switch based on the hardware connected and the target type currently in use. This will enable all errata workarounds for the determined silicon revision.

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` switch (on page 1-68). For example,

```
ccblkfn -proc ADSP-BF535 -si-revision 0.1 prog.c
```

If silicon version `none` is used, then no errata workarounds are enabled, whereas specifying silicon version `any` will enable all errata workarounds 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 and any errata workarounds which are appropriate for the latest silicon revision will be enabled.

The directory `Blackfin\lib` contains two sets of libraries: one set (suffixed with "`y`", for example, `libc532y.dlb`) contains workarounds for all known errata in all silicon revisions; the other set is built without any errata workarounds. Within the lib subdirectory, there are library directories for each silicon revision; these libraries have been built with errata workarounds appropriate for the silicon revision enabled. Note that an individual set of libraries may cover more than one specific silicon revision, so if several silicon revisions are affected by the same errata, then one common set of libraries might be used.

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`.

If the silicon revision is set to `any`, the `__SILICON_REVISION__` macro is set to `0xffff`. If the `-si-revision` switch is set to `none`, the compiler will not set the `__SILICON_REVISION__` macro.

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

Visit `http://www.analog.com/processors/technicalSupport/ICAnomalies.html` for information on specific anomalies (including anomaly IDs).

Using the -workaround Switch

The `-workaround` *`workaround_id`* switch (on page 1-81) 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 `<vdsp-anomaly-dictionary>` element.

The two main anomaly `.xml` files relevant to Blackfin processors are:

- `BLACKFIN-FRIO-anomaly.xml` - Applicable to the ADSP-BF535 processor
- `BLACKFIN-EDN-anomaly.xml` - Applicable to all other Blackfin processors

> 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 (on page 1-52); the memory-mapped register access functions (on page 1-275) 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" on page 1-102.

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 `-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 silicon revision is not `none` or 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.

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 draft document *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. VisualDSP++ 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 VisualDSP++ documentation.

The formats of the fixed-point types are given in table Table 1-15. 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 1-15. 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. VisualDSP++ 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" on page 3-330. 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 Table 1-16 for details of the suffixes and maximum and minimum fixed-point values.

Table 1-16. 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

Table 1-16. Fixed-Point Type Constant Suffixes and Macros (Cont'd)

Type	Suffix	Example	Minimum Value	Maximum Value
`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 might you write this 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:

Example

```
#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" on page 1-115, 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" on page 1-115.)

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" on page 1-110.

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" on page 1-118.)

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:

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 like

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

will not assign `0.5` to `f`, but will instead result 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}.

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.

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

```
#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 will emit 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" on page 1-112.

Compiler warnings will be 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 `bits`*`fx`* 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
}
```

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” on page 3-95.

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” on page 3-180.

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 on page 1-105, fixed-point operations that overflow give results saturated at the highest or lowest fixed-point value. Rounding is discussed in “Rounding Behavior” on page 1-118.

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" on page 1-108) contained the accumulation:

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

where sum was an `accum` type and `a[i]`, `b[i]` were `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];
```

will result in a multiply-accumulate instruction, but

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

- or -

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

will 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 also "#pragma FX_CONTRACT {ON|OFF}" on page 1-299) or with command-line switches, `-fx-contract` and `-no-fx-contract` (see "-fx-contract" on page 1-41 and "-no-fx-contract" on page 1-56). The pragma may be used at file scope or within functions. It obeys the same scope rules as the `FX_ROUNDING_MODE` pragma discussed on page 1-128 with an example in Listing 1-1 on page 1-129.

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 `round`*`fx`*, `strtofx`*`fx`*, and *`fx`*`divi` 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" on page 1-128.

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 `divi`*fx*, `idiv`*fx*, *fx*`divi`, `muli`*fx*, `abs`*fx*, `round`*fx*, `countls`*fx*, and `strtofx`*fx* families of functions.

divifx

The `divi`*fx* 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 will store the value 4 into the variable `quo`.

For more information, see "divifx" on page 3-130.

idivfx

The `idiv`*`fx`* 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 will store the value 2 into the variable `quo`.

For more information, see "idivfx" on page 3-207.

fxdivi

The *fx*`divi` 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 will store the value 0.5 into the variable `quo`.

For more information, see "fxdivi" on page 3-182.

mulifx

The `muli`*`fx`* 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 "mulifx" on page 3-249.

absfx

The `abs`*`fx`* 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 "absfx" on page 3-67.

roundfx

The `round`*`fx`* 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" on page 1-118 for more 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 fixed-point types.

For more information, see "roundfx" on page 3-280.

countlsfx

The `countls`*`fx`* 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 fixed-point types.

For more information, see "countlsfx" on page 3-113.

strtofxfx

The `strtofx`*fx* 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 "strtofxfx" on page 3-330.

I/O Conversion Specifiers

The `printf` and `scanf` families of functions support conversion specifiers for the fixed-point types. These are given in Table 1-17. 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 1-17. I/O Conversion Specifiers for the 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 "strtofxfx" on page 3-330.)

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 will have no effect unless either `-fixed-point-io` or `-full-io` are used. For more information, see "fprintf" on page 3-154.

Setting the Rounding Mode

As discussed in "Rounding Behavior" on page 1-118, 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` (on page 1-41)
- `-fx-rounding-mode-biased` (on page 1-41)
- `-fx-rounding-mode-unbiased` (on page 1-41)

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 will return to the outer scope rounding mode on exit from the compound statement. An example of how to use these pragmas is given in Listing 1-1.

Listing 1-1. 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}" on page 1-299 and "Changing the RND_MOD Bit" on page 1-242.

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 Table 1-18 on page 1-132.

Table 1-18. 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 1-18. Correspondence Between `fract16` and `fract32` Built-In Functions and Native Fixed-Point Arithmetic (Cont'd)

`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 could 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 would be 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 Table 1-19 on page 1-135 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 1-19. 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 Table 1-20 on page 1-136.

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 will not need to be changed.

Table 1-20. 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.

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 will be 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 Listing 1-2.

Listing 1-2. 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" on page 1-128.

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 `fract`s 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, and the ISO/IEC 14882:2003 C++ standard.

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" on page 1-156) that are compatible with the correct translation of standard-conforming programs. This is the default mode; it can be explicitly enabled by using the `-c89` switch (See "-c89" on page 1-26).

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" on page 1-156) that are compatible with the correct translation of standard-conforming programs. The compiler does not support the C99 keywords `_Complex` and `_Imaginary`. The ISO/IEC 9899:1990 C standard library provided in C89 mode is used in C99 mode. To enable C99 mode, use the `-c99` switch (See "-c99" on page 1-26).

In C mode, the best standard conformance is achieved using the default switches and the following non-default switches:

- `-const-strings` (See "-const-strings" on page 1-32)
- `-double-size-64` (See "-double-size-{32 | 64}" on page 1-34)
- `-full-io` (See "-full-io" on page 1-40)
- `-ieee-fp` (See "-ieee-fp" on page 1-45)
- `-decls-weak` (See "-decls-{weak|strong}" on page 1-33)
- `-enum-is-int` (See "-enum-is-int" on page 1-36)

The language extensions cannot be disabled to ensure strict compliance to the language standards. However, when compiling for MISRA-C ("MISRA-C Compiler Overview" on page 1-143) compliance checking, language extensions are disabled.

When the `-c89` switch is enabled (which is the default mode), 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.
- 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 freestanding implementation of the ISO/IEC 14882:2003 C++ standard, but it does not prohibit the use of language extensions ("C/C++ Compiler Language Extensions" on page 1-156) that are compatible with the correct translation of standard-conforming programs. 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.

In C++ mode, the best possible standard conformance is achieved using the following switches:

- `-no-anach` (See "-no-anach" on page 1-89)
- `-no-friend-injection` (See "-no-friend-injection" on page 1-89)
- `-no-implicit-inclusion` (See "-no-implicit-inclusion" on page 1-89)
- `-std-templates` (See "-std-templates" on page 1-90)
- `-const-strings` (See "-const-strings" on page 1-32)
- `-double-size-64` (See "-double-size-{32 | 64}" on page 1-34)
- `-eh` (See "-eh" on page 1-35)
- `-extern-inline` (See "-extern-inline" on page 1-87)
- `-full-io` (See "-full-io" on page 1-40)
- `-ieee-fp` (See "-ieee-fp" on page 1-45)

- `-decls-weak` (See "-decls-{weak|strong}" on page 1-33)
- `-rtti` (See "-rtti" on page 1-90)

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 "Rules Descriptions" on page 1-147. 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 pragmas "#pragma section/#pragma default_section" on page 1-310 and "#pragma inline" on page 1-320. Rules can be suppressed by the use of command-line switches or the MISRA extensions to "Diagnostic Control Pragmas" (on page 1-338).

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.

Refer to Table 1-6 on page 1-24 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 VisualDSP++ compiler is one of the most comprehensive MISRA-C:2004 compliance checking tools available. The compiler provides command-line switches (on page 1-83) and diagnostic control pragmas (on page 1-338) 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 focussing 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 */
}
```

Rules Descriptions

The following are brief explanations of how some of the MISRA-C rules are supported and interpreted in this VisualDSP++ 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" on page 1-448.

- **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 (on page 1-84).

- **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 (on page 1-84).

- **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 (on page 1-84).

- **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 <misra_types.h> and <stdbool.h>. The rule is only enforced when the `-misra-strict` compiler switch is enabled (on page 1-84).

- **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 (on page 1-84).

- **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 (on page 1-84).

- **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.

- **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 (on page 1-84).

- **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 (on page 1-84).

- **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. The run-time code is not added if the `-misra-no-runtime` compiler switch is enabled (on page 1-84).

- **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 (on page 1-84).

- **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 the `-misra-strict` compiler switch is enabled (on page 1-84).

- **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 (on page 1-84).

- **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 (on page 1-84).

- **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 (on page 1-84), 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 (on page 1-84).

- **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.

- **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 redeclarations 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 (on page 1-84).

C/C++ Compiler Language Extensions

The compiler supports extensions to the ANSI/ISO standards for the C and C++ languages. These extensions add support for DSP hardware and permit some C++ programming features when compiling in C mode. Most extensions are also available when compiling in C++ mode.

This section contains information on ISO/IEC 9899:1999 standard features that are supported in C89 mode:

- "Function Inlining" on page 1-159
- "Variable Argument Macros" on page 1-164
- "Restricted Pointers" on page 1-165
- "Variable-Length Arrays" on page 1-166
- "Non-Constant Initializer Support" on page 1-167
- "Designated Initializers" on page 1-168
- "Hexadecimal Floating-Point Numbers" on page 1-170
- "Declarations Mixed With Code" on page 1-171
- "Compound Literals" on page 1-172
- "C++ Style Comments" on page 1-173
- "Enumeration Constants That Are Not int Type" on page 1-173
- "Boolean Type Support Keywords (bool, true, false)" on page 1-173

This section also contains information on other language extensions:

- "Native Fixed-Point Types fract and accum" on page 1-174
- "Inline Assembly Language Support Keyword (asm)" on page 1-174
- "Bank Qualifiers" on page 1-191
- "Placement Support Keyword (section)" on page 1-192
- "Placement of Compiler-Generated Code and Data" on page 1-193
- "Long Identifiers" on page 1-194
- "Compiler Built-In Functions" on page 1-195
- "Pragmas" on page 1-277
- "GCC Compatibility Extensions" on page 1-349
- "Preprocessor-Generated Warnings" on page 1-357

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 beginning on page 1-26.

> This section describes the shorter forms of the keyword extensions. In most cases, you can use either form in your code. For example, all references to the `inline` keyword in this text appear without the leading double underscores, but you can interchange `inline` and `__inline` in your code.

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`.

Table 1-21 and Table 1-22 provide descriptions of each extension and direct you to sections that describe each extension in more detail.

Table 1-21. 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" on page 1-159.
`asm()`	Places Blackfin core assembly language commands directly in your C/C++ program. For more information, see "Inline Assembly Language Support Keyword (asm)" on page 1-174.
`bank("string")`	Specifies a name which the user assigns to associate declarations that reside in particular memory banks. For more information, see "Bank Qualifiers" on page 1-191.
`section("string")`	Specifies the section in which an object or function is placed. For more information, see "Placement Support Keyword (section)" on page 1-192.

`bool`
`true`
`false`
	Specifies a Boolean type. For more information, see "Boolean Type Support Keywords (bool, true, false)" on page 1-173.

`restrict`	Specifies restricted pointer features. For more information, see "Restricted Pointers" on page 1-165.

Table 1-22. 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" on page 1-167.
Indexed initializers	Specifies elements of an aggregate initializer in arbitrary order. For more information, see "Designated Initializers" on page 1-168.
Variable-length arrays	Creates local arrays with a variable size. For more information, see "Variable-Length Arrays" on page 1-166.
Long identifiers	Supports identifiers of up to 1022 characters in length. For more information, see "Long Identifiers" on page 1-194.
Preprocessor-generated warnings	Generates warning messages from the preprocessor. For more information, see "Preprocessor-Generated Warnings" on page 1-357.
C++ style comments	Allows for "`//`" C++ style comments in C programs. For more information, see "C++ Style Comments" on page 1-173.

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" on page 1-26.

This keyword 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 needs to 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` 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{error|remark|suppress|warn}" on page 1-79.

Function inlining can also occur by use of the `-Oa` (automatic function inlining) switch ("-Oa" on page 1-60), 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 (on page 1-51), the compiler will not inline the call. If the call is to a function that has `#pragma always_inline` specified (see "Inline Control Pragmas" on page 1-301), a warning will also be issued.

3. If the call is to a function that has `#pragma never_inline` specified, the call will not be 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 will not be 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 has `#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 `#pragma 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" on page 1-61). 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).

The inlining process also considers the required stack size while inlining. A function that has a local array of 20 integers needs such an array for each inlined invocation, and if inlined many times, the cumulative effect on overall stack requirements can be significant. Consequently, the compiler considers both the stack space required by the called function, and the total stack space required by the caller; either may reach a limit at which the compiler determines that inlining the call would not be beneficial. The stack size analysis is not subject to the `-Ov` switch.

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

If a function is static (that is, private to the module being compiled) and all calls to that function are inlined, there are no calls remaining that are not inline. Consequently, the compiler does not generate an out-of-line copy for the function, thus reducing the size of the resulting application.

If the address of the function is taken, it is possible that the function could be called through that derived pointer, so the compiler cannot guarantee that all calls have been accounted for. In such cases, an out-of-line copy is generated.

A function declared `inline` must be defined (its body must be included) in every file in which the function is used. This is normally done by placing the `inline` definition in a header file. Usually it is also declared static.

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 (on page 1-301), however, or the -always-inline switch (on page 1-29) is in force, the compiler will inline the call despite the mismatch in sections.

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 ... to indicate the parameter stands for a variable number of arguments.

For example:

```
#define trace(file,line,...) \
        logmsg(file,line,__VA_ARGS__)
```

can be used with differing numbers of arguments:

```
trace("a.c", 22, "Got here!\n”);
```

```
trace("b.c", 99, "i = %d\n", i);

trace("c.c", 72, "x = %f, y = %f\n", x, y);
```

This variable argument macro syntax comes from the ISO/IEC 9899:1999 C standard. The compiler supports both GCC and C99 variable argument macro formats in C89, C99, and C++ modes. (See "GCC Variable Argument Macros" on page 1-353)

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. This keyword specifies that the pointer provides exclusive initial access to the pointed object. 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 that 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 your 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. (For more information, see “-c89” on page 1-26.) Variable-length arrays are not supported in C++ mode.

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.

Variable-length arrays are only supported as an extension to C; variable-length arrays are not supported in C++.

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 deallocates the array when this occurs.

You can use variable-length arrays as function arguments, such as:

```
struct entry
   var_array (int array_len, char data[array_len][array_len])
   {
   ...
   }
```

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 much easier than explicitly allocating it in a heap.

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.

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 `binary-exponent-part` 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 causes `f` to be initialized with the value `0x800000`.

```
float f = 0x1p-126f;
```

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

This is a standard feature of the ISO/IEC:9899 1999 standard. The compiler supports it as an extension in C89 mode. It is not allowed in C++ mode.

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 VisualDSP++ compiler allows enumeration constants to be integer types other than `int`, such as `unsigned int`, `long long` or `unsigned long long`, if the enumeration constant has a value outside the range of `int`.

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 has not been 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. This support is available for the C language only. A discussion of how to use this support is given in "Using Native Fixed-Point Types" on page 1-104.

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.

> 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.

> `asm()` constructs with inputs, outputs or affected registers are executable statements, and as such, may not appear before declarations within C/C++ functions. The `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.

> 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.

> The compiler generates a label before and after inline assembly instructions when generating debug code. (See the `-g` switch on page 1-42.) 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:

- "asm() Construct Syntax" on page 1-176
- "Assembly Construct Operand Description" on page 1-180
- "Using long long Types in asm Constraints" on page 1-185
- "Assembly Constructs With Multiple Instructions" on page 1-186
- "Assembly Construct Reordering and Optimization" on page 1-187
- "Assembly Constructs With Input and Output Operands" on page 1-188
- "Assembly Constructs With Compile-Time Constants" on page 1-189
- "Assembly Constructs and Flow Control" on page 1-190
- "Guidelines for Using asm() Statements" on page 1-190

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" on page 1-186.

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" on page 1-180.

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 Table 1-24 on page 1-185.

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 writes to 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`).

> 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.

Table 1-23 on page 1-183 describes the correspondence between constraint letters and register classes.

> The use of any letter not listed in Table 1-23 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 `=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`

Table 1-23 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. Table 1-24 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 Table 1-24.

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  */
);
```

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. The clobber string can be any of the registers recognized by the compiler.

Table 1-23. 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`

Table 1-23. asm() Operand Constraints (Cont'd)

Constraint	Register Type	Registers
`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`
`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" on page 1-189.)	
`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 1-24. 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;

int main(void) {
    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" on page 1-406.

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  */
```

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 Table 1-24.)

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"
```

produces code like

```
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"
```

produces code like

```
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

> Do not place flow-control operations within an `asm()` construct that "leaves" the `asm()` construct functions, such as calling a procedure or performing a jump to another piece of code that is not within the `asm()` construct itself. Such operations are invisible to the compiler, may result in multiple-defined symbols, and may violate assumptions made by the compiler.

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" on page 1-259).

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 VisualDSP++ (see also "Memory-Mapped Register Access Built-In Functions" on page 1-275).

Bank Qualifiers

Bank qualifiers can be attached to data declarations to indicate that the data resides in particular memory banks. For example,

```
int bank("blue")  *ptr1;

int bank("green") *ptr2;
```

The `bank` qualifier assists the optimizer because the compiler assumes that if two data items are in different banks, they can be accessed together without conflict.

The bank name string literals have no significance, except to differentiate between banks. There is no interpretation of the names attached to banks, which can be any arbitrary string. There is a current implementation limit of ten different banks.

For any given function, three banks are defined automatically. These are:

- The default bank for global data.
  The "static" or "extern" data that is not explicitly placed into another bank is assumed to be within this bank. Normally, this bank is called "`__data`", although a different bank can be selected with `#pragma data_bank(`*`bankname`*`)`.
- The default bank for local data.
  Local variables of "auto" storage class that are not explicitly placed into another bank are assumed to be within this bank. Normally, this bank is called "`__stack`", although a different bank can be selected with `#pragma stack_bank(`*`bankname`*`)`.
- The default bank for the function's instructions.
  The function itself is placed into this bank. Normally, it is called "`__code`", although a different bank can be selected with `#pragma code_bank(`*`bankname`*`)`.

Each memory bank can have different performance characteristics. For more information on memory bank attributes, see "Memory Bank Pragmas" on page 1-341.

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 "Using Memory Sections" on page 1-422.

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 declaration 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" on page 1-310 for more information.

> 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 (on page 1-72) can be used to specify alternatives for these defaults on the command-line, and the "#pragma section/#pragma default_section" on page 1-310 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 code 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++
```

> It is the user's responsibility to ensure that appropriately named sections exist in the `.ldf` file.

The compiler currently supports the following section identifiers:

`code`	Controls placement of machine instructions. Default is `program`.
`data`	Controls placement of initialized variable data. Default is `data1`.

`constdata`	Controls placement of constant data. Default is `constdata`.
`bsz`	Controls placement of zero-initialized variable data. Default is `bsz`.
`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” on page 1-419.
`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. Default is `vtbl`.
`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

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 intrinsic (built-in) functions that enable efficient use of hardware resources. These functions are:

- "Video Operation Built-In Functions" on page 1-267
- "Misaligned Data Built-In Functions" on page 1-274
- "Memory-Mapped Register Access Built-In Functions" on page 1-275
- "Miscellaneous Built-In Functions" on page 1-276

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. The header files also define more readable names for the built-in functions without the `__builtin_` prefix. These additional names are disabled if the `-no-builtin` command-line switch is used.

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.

Fractional Value Built-In Functions in C

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" on page 1-104. Alternatively, built-in functions may be used to specify fractional operations. This section discussed the use of these built-in functions.

Table 1-25. Fractional Value C Types

C type	Usage
`fract16`	Single 16-bit signed fractional value, typedef to short
`fract32`	Single 32-bit signed fractional value, typedef to 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

The various C types used in the built-in functions described in this section are described in Table 1-25.

See "Using Data Storage Formats" on page 1-443 for information on how `fract16`, `fract32`, `fract`, `long fract`, and `fract2x16` types are represented. See the *Blackfin Processor Programming Reference* 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.

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" on page 1-198 for descriptions of built-in functions that work primarily with fract16 data. See "32-Bit Fractional Built-In Functions" on page 1-203 for descriptions of built-in functions that work primarily with fract32 data.

See "fract2x16 Built-In Functions" on page 1-207 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, recoding 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 Built-In Functions" on page 1-215 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.

```
int norm_fr1x16(fract16 f1)
int 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, multiplies 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: `{a<<shft,b<<shft}`

`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 `fract16s` 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 `fract16s` 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 `fract16s` 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 `fract16s` 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 `fract16s` 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 `fract16s` 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 `fract16s` 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(fract2x16 f1)
fract diff_hl_fx_fr2x16(fract2x16 f1)
```

Takes the difference (high-low) of the two `fract16`s or `fract`s 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 `fract16`s or `fract`s 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 Built-In Functions

If `fract.h` is included with `ETSI_SOURCE` defined, the macros listed below are also defined, mapping from the European Telecommunications Standards Institute (ETSI) `fract` functions onto the compiler built-in functions. The mappings are done in `fract_math.h` (included by `fract.h`).

```
add()                          abs_s()
sub()                          saturate()
shl()                          extract_h()
shr()                          extract_l()
mult()                         L_deposit_l()
mult_r()                       div_s()
negate()                       norm_s()
round()                        norm_l()
```

`L_add()`	`L_Extract()`
`L_sub()`	`L_Comp()`
`L_abs()`	`Mpy_32()`
`L_negate()`	`Mpy_32_16()`
`L_shl()`	`L_mult()`
`L_shr()`	`L_mac()`
`L_msu()`	`L_shr_r()`
`div_l()`	

Here is a description of the ETSI functions that do not map exactly to compiler built-in functions:

```
fract32 L_mac(fract32 acc,fract16 f1, fract16 f2)
```

Multiply accumulate. Returns `acc+=f1*f2`.

```
fract32 L_msu(fract32 acc,fract16 f1, fract16 f2)
```

Multiply subtract. Returns `acc-=f1*f2`.

```
fract32 L_Comp(fract16 f1, fract16 f2)
```

Composes a 32-bit value from the given high and low components. The sign is provided with the low half, and the result is calculated as: `f1<<16 + f2<<1`.

```
fract32 Mpy_32_16(short hi, short lo, fract16 n)
```

Multiplies a `fract32` (decomposed to `hi` and `lo`) by a `fract16`, and returns the result as a `fract32`.

```
void L_Extract(fract32 f1, fract16 *f2, fract16 *f3)
```

Decomposes a 32-bit `fract` into two 16-bit `fracts`.

```
fract32 Mpy_32(short hi1, short lo1, short hi2, short lo2)
```

Multiplies two `fract32` numbers, and returns the result as a `fract32`. The input `fracts` have both been split up into two `shorts`.

```
fract16 div_s(fract16 f1, fract16 f2)
```

Produces a result which is the fractional division of `f1` by `f2`. Not a built-in function as written in C code.

By default, the following ETSI functions map to clipping versions of the built-in `fract` shifts.

```
fract16 shl(fract16 _x, short _y);
fract16 shr(fract16 _x, short _y);
fract32 L_shl(fract32 _x, short _y);
fract32 L_shr(fract32 _x, short _y);
```

To map them to the faster, non-clipping, versions of the built-in fractional shifts, define the macro `_ADI_FAST_ETSI` in your source before you include `fract_math.h` or on the compile command line.

ETSI Support

VisualDSP++ 5.0 provides ETSI support routines in the `libetsi*.dlb` library, which contains routines for manipulation of the `fract16` and `fract32` data types as stipulated by ETSI. The routines provide bit-accurate calculations for common operations, and conversions between `fract16` and `fract32` data types.

To use the ETSI routines, the header file `libetsi.h` must be included, and all source code must be compiled with the `ETSI_SOURCE` macro defined.

These routines are:

- "32-Bit Fractional ETSI Routines Using Double-Precision Format" on page 1-220
- "32-Bit Fractional ETSI Routines Using 1.31 Format" on page 1-223
- "16-Bit Fractional ETSI Routines" on page 1-227

Several of the ETSI routines are provided with carry and overflow checking. Where overflow or carry occurs, the global variables `Carry` and `Overflow` are set. It is your responsibility to reset these variables in between operations.

The `Carry` and `Overflow` variables are represented by integers and are prototyped in the `libetsi.h` system header file.

Two types of `libetsi` libraries are provided with VisualDSP++ 5.0:

- Those with a name of the form `libetsi*co.dlb` have been compiled with checking and setting of `Overflow` and `Carry`.
- Those with a name of the form `libetsi*.dlb` (with no "`co`") have the checking and setting of `Overflow` and `Carry` disabled for optimal performance. To use the `Carry` and `Overflow` checking versions of the library, use the compiler flag "`-l etsi*co`". When rebuilding `libetsi`, `Carry` and `Overflow` checking is enabled with the C and assembler macro definition `__SET_ETSI_FLAGS=1`.

By default, the carry/overflow setting function libraries (`libetsi*co.dlb`) are not built by the supplied makefiles. To rebuild the carry and overflow setting versions of the libraries, define compiler macro `__SET_ETSI_FLAGS =1` during compilation.

The carry/overflow setting versions of the following functions will not set the `Carry` and/or `Overflow` variables correctly on the ADSP-BF535 processor, due to differences in the way the hardware flags are set on the ADSP-BF535 processor.

```
shl       shr      shr_r

L_msuNs   L_shl    L_shr    L_shr_r
```

Many routines in the library are also represented by built-in functions. Where built-in functions exist, the compiler replaces the functional code with an optimal inline assembler representation. To disable the use of the ETSI built-in functions and use the library versions, compile with the macro `NO_ETSI_BUILTINS` defined. However, use of the built-in functions results in better performance since there is an overhead in making the function call to the library.

The built-in versions of the functions do not set the `Carry` and `Overflow` flags.

The built-in versions of some ETSI functions are 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" on page 1-242.

If the macro `RENAME_ETSI_NEGATE` is defined, the ETSI function "`negate`" will be renamed to `etsi_negate()`. This is useful because the C++ Standard declares a template function called `negate()` (found in the C++ include "`functional`").

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
- 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:

```
L_32 = (hi<<16) + (lo<<1)
```

where:

- L_32 is a 32-bit signed integer (though it is listed as fract32)
- hi and lo are 16-bit signed integers (though they are listed as fract16)

The ETSI operations that use DPF are:

```
fract32 L_Comp(fract16 hi, fract16 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);
```

A built-in version of this function is also provided.

`void L_Extract(fract32 src, fract16 *hi, fract16 *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
```

A built-in version of this function is also provided.

`fract32 Mpy_32(fract16 hi1,fract16 lo1, fract16 hi2,fract16 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);
```

A built-in version of this function is also provided.

`fract32 Mpy_32_16(fract16 hi, fract16 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. A built-in version of this function is also provided.

`fract32 Div_32(fract32 L_num, fract16 denom_hi, fract16 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()`).

```
#define ETSI_SOURCE
#include <libetsi.h>
fract32 mul32by32_etsi(fract32 a, fract32 b) {
   fract32 exp_prec_res;
   fract16 lo1, hi1, lo2, hi2, hi, lo;
   fract32 res;

   /* Extract two 16-bit DPF components from a 32-bit fract */
   L_Extract(a, &hi1, &lo1) ;

   /* Extract two 16-bit DPF components from a 32-bit fract */
   L_Extract(b, &hi2, &lo2) ;

   /* 32-bit extended precision Multiply */
   exp_prec_res = Mpy_32(hi1, lo1, hi2, lo2);

   /* Extract two 16-bit DPF components from a 32-bit integer */
   L_Extract(exp_prec_res, &hi, &lo);

   /* Compose a 32-bit integer from two 16-bit DPF components */
   res = L_Comp(hi, lo);
```

```
    /* 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. When using a version of the library compiled with `__SET_ETSI_FLAGS`, the `Carry` and `Overflow` flags are set when carry and overflow/underflow occur during addition.

`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. A built-in version of this function is also provided.

`fract32 L_add(fract32 a, fract32 b)`

Returns the 32-bit saturated result of the addition of the two input parameters. If the library is compiled with `__SET_ETSI_FLAGS`, the `Overflow` flag is set when overflow occurs. A built-in version of this function is also provided.

`fract32 L_deposit_h(fract16 hi)`

Deposits the 16-bit parameter into the 16 most significant bits of the 32-bit result. The least 16 bits are zeroed. A built-in version of this function is also provided.

`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 set to sign extension for the input. A built-in version of this function is also provided.

`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 multiplication result with the 32-bit parameter. A built-in version of this function is also provided.

`fract32 L_macNs(fract32,fract16, fract16)`

Performs a non-saturating version of the `L_mac` operation. If the library is compiled with `__SET_ETSI_FLAGS`, the `Overflow` and `Carry` flags are set when carry or overflow/underflow occurs.

`fract32 L_mls (fract32, fract16)`

Multiplies both the most significant bits and the least significant bits of a `long`, by the same `short`.

`fract32 L_msu(fract32, fract16, fract16)`

Performs a fractional multiplication of the two 16-bit parameters and returns the saturated subtraction of the multiplication result with the 32-bit parameter. A built-in version of this function is also provided.

`fract32 L_msuNs(fract32, fract16, fract16)`

Performs a non-saturating version of the `L_msu` operation. If the library is compiled with `__SET_ETSI_FLAGS`, the `Overflow` and `Carry` flags are set when carry or overflow/underflow occurs.

`fract32 L_mult(fract16, fract16)`

Returns the 32-bit saturated result of the fractional multiplication of the two 16-bit parameters. A built-in version of this function is also provided.

`fract32 L_negate(fract32)`

Returns the 32-bit result of the negation of the parameter. Where the input parameter is `0x80000000` saturation occurs and `0x7fffffff` is returned. A built-in version of this function is also provided.

`fract32 L_sat(fract32)`

The resultant variable is set to `0x80000000` if `Carry` and `Overflow` flags are set (underflow condition); else, if `Overflow` is set, the resultant is set to `0x7fffffff`. The default revision of the library simply returns as no checking or setting of the `Overflow` and `Carry` flags is performed.

`fract32 L_shl(fract32 src, fract16 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`, sign-extended. The result is saturated in cases of overflow and underflow.

If the library is compiled with `__SET_ETSI_FLAGS`, the `Overflow` flag is set when overflow occurs. A built-in version of this function is also provided.

`fract32 L_shr(fract32, fract16)`

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.

If the library is compiled with `__SET_ETSI_FLAGS`, the `Overflow` flag is set when overflow occurs. A built-in version of this function is also provided.

`fract32 L_shr_r(fract32, fract16)`

Performs the shift-right operation as per `L_shr` but with rounding. If the library is compiled with `__SET_ETSI_FLAGS`, the `Overflow` and `Carry` flags are set when carry or overflow/underflow occurs.

`fract32 L_sub(fract32, fract32)`

Returns the 32-bit saturated result of the subtraction of two 32-bit parameters (first-second). A built-in version of this function is also provided.

```
fract32 L_sub_c(fract32 f1, fract32 f2)
```

Performs 32-bit subtraction of two fractional values (`f1 - f2`). When using a version of the library compiled with `__SET_ETSI_FLAGS`, the `Carry` and `Overflow` flags are set when carry and overflow/underflow occur during subtraction.

16-Bit Fractional ETSI Routines

The following functions return or primarily operate on 16-bit fractional data.

```
fract16 abs_s(fract16)
```

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. A built-in version of this function is also provided.

```
fract16 add(fract16, fract16)
```

Returns the 16-bit result of adding the two `fract16` input parameters.

Saturation occurs with the result being set to `0x7fff` for overflow and `0x8000` for underflow. If the library is compiled with `__SET_ETSI_FLAGS`, the `Overflow` and `Carry` flags are set when carry or overflow/underflow occurs. A built-in version of this function is also provided.

```
fract16 div_l (fract32, fract16)
```

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`. A built-in version of this function is also provided.

`fract16 extract_l(fract32)`

Returns the 16 least significant bits of the 32-bit fract parameter provided. A built-in version of this function is also available.

`fract16 extract_h(fract32)`

Returns the 16 most significant bits of the 32-bit fract parameter provided. A built-in version of this function is also available.

`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, fract16, fract16)`

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, fract16)`

Returns the 16-bit result of the fractional multiplication of the input parameters. The result is saturated. A built-in version of this function is also provided.

`fract16 mult_r(fract16, fract16)`

Performs a 16-bit multiply with rounding of the result of the fractional multiplication of the two input parameters. A built-in version of this function is also provided.

The inline version of the `mult_r()` function is implemented using the `multr_fr1x16()` compiler intrinsic, which in turn does a normal 16-bit fractional multiply:

```
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 are not always ETSI-compliant. To avoid this issue, set `RND_MOD` before using the inline version or use the `libetsi` library-defined version of the function (which sets the bit). For more information, see "Changing the RND_MOD Bit" on page 1-242.

`fract16 negate(fract16)`

Returns the 16-bit result of the negation of the input parameter. If the input is `0x8000`, saturation occurs and `0x7fff` is returned. A built-in version of this function is also provided.

This function generates the Blackfin `SIGNBITS` instruction.

`int norm_l(fract32)`

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`.

`int norm_s(fract16)`

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`.

This function generates the Blackfin `SIGNBITS` instruction.

`fract16 round(fract32)`

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. A built-in version of this function is also provided.

`fract16 saturate(fract32)`

Returns the 16 least significant bits of the input parameter. If the input parameter is greater than `0x7fff`, `0x7fff` is returned. If the input parameter is less than `0x8000`, `0x8000` is returned. A built-in version of this function is also available.

`fract16 shl(fract16 src, fract16 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.

If the library is compiled with `__SET_ETSI_FLAGS`, the `Overflow` and `Carry` flags are set when carry or overflow/underflow occurs. A built-in version of this function is also provided.

`fract16 shr(fract16, fract16)`

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.

If the library is compiled with `__SET_ETSI_FLAGS`, the `Overflow` and `Carry` flags are set when carry or overflow/underflow occurs. A built-in version of this function is also provided.

`fract16 shr_r(fract16, fract16)`

Performs a shift to the right as per the `shr()` operation with additional rounding and saturation of the result.

`fract16 sub(fract16 f1, fract16 f2)`

Returns the 16-bit result of the subtraction of the two parameters (`f1 - f2`). Saturation occurs with the result being set to `0x7fff` for overflow and `0x8000` for underflow.

If the library is compiled with `__SET_ETSI_FLAGS`, the `Overflow` and `Carry` flags are set when carry or overflow/underflow occurs. A built-in version of this function is also provided.

Fractional Value Built-In Functions in C++

The compiler provides support for two C++ fractional classes. The `fract` class uses a `fract32` C type for storage of the fractional value, whereas the `shortfract` class uses a `fract16` C type for storage of the fractional value.

Instances of the `shortfract` and `fract` classes are initialized using values with the "`r`" suffix, provided they are within the range `[-1,1)`. The `fract` class is implemented by the compiler as representing the internal type `fract`. For example,

```
#include <fract>
int main ()
{
   fract X = 0.5r;
}
```

Instances of the `shortfract` class can be initialized using "`r`" values in the same way, but are not represented as an internal type by the compiler. Instead, the compiler produces a temporary `fract`, which is initialized using the "`r`" value. The value of the `fract` class is then copied to the `shortfract` class using an implicit copy, and the `fract` is destroyed.

The `fract` and `shortfract` classes contain routines that allow basic arithmetic operations and movement of data to and from other data types. The example below shows the use of the `shortfract` class with `*` and `+` operators.

The mathematical routines for addition, subtraction, division, and multiplication for both `fract` and `shortfract` classes are performed using the ETSI-defined routines for the C fractional types (`fract16` and `fract32`). Inclusion of the `fract` and `shortfract` header files implicitly defines the macro `ETSI_SOURCE` to be 1. This is required for use of the ETSI routines, which are defined in `libetsi.h` and located in the `libetsi53*.dlb` libraries.

```
#include <shortfract>
#include <stdio.h>
#define N 20

shortfract x[N] = {
   .5r,.5r,.5r,.5r,.5r,
   .5r,.5r,.5r,.5r,.5r,
   .5r,.5r,.5r,.5r,.5r,
   .5r,.5r,.5r,.5r,.5r};

shortfract y[N] = {
   0,.1r,.2r,.3r,.4r,
   .5r,.6r,.7r,.8r,.9r,
   .10r,.1r,.2r,.3r,.4r,
   .5r,.6r,.7r,.8r,.9r};

shortfract fdot(int n, shortfract *x, shortfract *y)
{
   int j;
   shortfract s;
   s = 0;
   for (j=0; j<n; j++) {
         s += x[j] * y[j];
   }
   return s;
}
```

```
int main(void)
{
   fdot(N,x,y);
}
```

fract16 and fract32 Literal Values in C

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” on page 1-107.

When compiling a program in C mode, 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 will yield an unexpected result (see “Data Type Conversions and Fixed-Point Types” on page 1-110 for more details). However, in C++ mode the “`r`” suffix denotes values of the `fract` class. If a C program is compiled in C++ mode, `fract16` and `fract32` variables can be initialized using “`r`” literal values; the compiler automatically converts from the `fract` class values to the C types. When adopting this approach, be aware of any semantic differences between the C and C++ languages that might affect your program.

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`.

> 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 VisualDSP++ 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:

```
fract32 fr16_to_fr32(fract16); // Deposits a fract16 to make
                               // a fract32
fract16 fr32_to_fr16(fract32); // Truncates a fract32 to make
                               // a fract16

fract32 float_to_fr32(float);  // Converts a float to fract32
fract16 float_to_fr16(float);  // Converts a float to fract16

float fr16_to_float(fract16);  // Converts a fract16 to float
float fr32_to_float(fract32);  // Converts a fract32 to float
```

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.

```
long fract fx16_to_fx32(fract); // Deposits a fract to make
                                // a long fract
fract fx32_to_fx16(long fract); // Truncates a long fract to make
                                // a fract

long fract float_to_fx32(float);   // Converts a float
```

```
                                    // to a long fract
fract float_to_fx16(float);         // Converts a float to a fract

float fx16_to_float(fract);         // Converts a fract to a float
float fx32_to_float(long fract);    // Converts a long fract
                                    // to a float
```

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`.

Listing 1-3 implements a floating-point multiplication using an ETSI `fract` implementation.

Listing 1-3. Floating-Point Multiplication Using fracts

```
#define ETSI_SOURCE
#include <fract2float_conv.h>
#include <fract_typedef.h>
#include <libetsi.h>
#include <stdlib.h>
#include <math.h>

/* return a*b calculated using fract implementation */
float mul_fp(float a, float b) {
   int sign_a, sign_b, sign_res;
   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;
fract16 hia, loa, hib, lob;

/* if either input is 0, return 0 */
if (a == 0.0 || b == 0.0)
   return 0.0;

/* get sign and take absolute of inputs */
if (*(unsigned int *)&a & 0x80000000) {
   sign_a=-1;
   a = fabs(a);
} else
   sign_a=1;

if (*(unsigned int *)&b & 0x80000000) {
   sign_b=-1;
   b = fabs(b);
} else
   sign_b=1;

/* compute sign of result */
sign_res = sign_a * sign_b;

/* 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);

    /* negate result if necessary */
    result = result * sign_res;
    /* return result */
    return result;
}  /* mul_fp */
```

Complex Fractional Built-In Functions in C

The `complex_fract16` type is used to hold complex fractional numbers. It contains real and imaginary values, both as 16-bit fractional numbers.

```
typedef struct {
    fract16 re, im;
} complex_fract16;
```

The `complex_fract32` type is used to hold complex fractional numbers. It contains real and imaginary values, both as 32-bit fractional numbers.

```
typedef struct {
    fract32 re, im;
} complex_fract32;
```

The `complex_fract16` and `complex_fract32` types are defined by the `complex.h` header file. Additionally, there are numerous library functions for manipulating complex fracts. These functions are documented in "DSP Run-Time Library Reference" on page 4-75.

The compiler also supports the following built-in operations for complex `fracts`. 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 perform a complex fractional multiplication of `_a` and `_b`, the result of which is accumulated with `_sum`, saturating the accumulation at 32 bits:

```
complex_fract16 cmac_fr16(complex_fract16 _sum,
                          complex_fract16 _a,
                          complex_fract16 _b);
```

- The following built-in function generates instructions to perform a complex fractional multiplication of `_a` and `_b`, the result of which is subtracted from `_sum`, saturating the result at 32 bits:

```
complex_fract16 cmsu_fr16(complex_fract16 _sum,
                          complex_fract16 _a,
                          complex_fract16 _b);
```

- The following built-in function generates instructions to calculate and returns the complex fractional square of `_a`.

```
complex_fract16 csqu_fr16(complex_fract16 _a);
```

- The following built-in functions generate instructions to calculate the square of the distance between inputs `_x` and `_y`.

```
fract16 cdst_fr16(complex_fract16 _x,
                  complex_fract16 _y);
fract32 cdst_fr32(complex_fract16 _x,
                  complex_fract16 _y);
fract cdst_fx_fr16(complex_fract16 _x,
                   complex_fract16 _y);
long fract cdst_fx_fr32(complex_fract16 _x,
                        complex_fract16 _y);
```

- Complex fractional multiply accumulate and complex fractional multiply subtract operations with internal operations performed saturating to 40-bits in the accumulator registers.

```
complex_fract16 cmac_fr16_s40(complex_fract16 _sum,
                              complex_fract16 _a,
                              complex_fract16 _b);
complex_fract16 cmsu_fr16_s40(complex_fract16 _sum,
                              complex_fract16 _a,
                              complex_fract16 _b);
```

- The following functions can be used to extract the real (`real_fr32`) and imaginary (`imag_fr32`) parts of the `complex_fract16` or `complex_fract32` input `_a`.

```
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.

```
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 function performs a complex addition of the inputs and returns the result.

```
complex_fract32 cadd_fr32(complex_fract32 _a,
complex_fract32 _b);
```

- The following function performs a complex subtraction of the inputs and returns the result.

```
complex_fract32 csub_fr32(complex_fract32 _a,
complex_fract32 _b);
```

- The following function returns the complex conjugate of the input.

```
complex_fract32 conj_fr32(complex_fract32 _a);
```

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 will mean 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 will mean that instructions that depend on the state of the `RND_MOD` bit will 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 you might 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}” on page 1-299 and “Setting the Rounding Mode” on page 1-128 for further details.

Complex Operations in C++

The C++ complex class is defined in the `<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.

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;

complex<double> inverse2d(const complex<double> mx[4],
complex<double> mxinv[4])
{
   complex<double> det = mx[0] * mx[3] - mx[2] * mx[1];

   if( (det.real() != 0.0) || (det.imag() != 0.0) ) {
      complex<double> invdet = complex<double>(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 add_i2x16(int2x16 _x, int2x16 _y);
int2x16 sub_i2x16(int2x16 _y, int2x16 _y);
int2x16 mult_i2x16(int2x16 _x, int2x16 _y);
int2x16 abs_i2x16(int2x16 _x);
int2x16 min_i2x16(int2x16 _x, int2x16 _y);
int2x16 max_i2x16(int2x16 _x, int2x16 _y);
```

The following function performs summation of the two packed components:

```
int sum_i2x16(int2x16 _x);
```

The following functions provide cross-wise multiplication:

```
int mult_ll_i2x16(int2x16 _x, int2x16 _y);
int mult_hl_i2x16(int2x16 _x, int2x16 _y);
int mult_lh_i2x16(int2x16 _x, int2x16 _y);
int mult_hh_i2x16(int2x16 _x, int2x16 _y);
```

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*.

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.

Listing 1-4 shows a dot product that is guaranteed to accumulate in 40-bits and to saturate the final sum to 32-bits.

Listing 1-4. Fractional Dot Product Implemented with Accumulator Built-Ins

```
#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);
}
```

Accumulator Built-In Function Prototypes

Table 1-26 lists 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. See the *Blackfin Processor Programming Reference* for details of the instructions.

Table 1-26. 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)

Table 1-26. Accumulator Built-In Functions (Cont'd)

Function	Instruction
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;
acc40 A_ashift(acc40, short);	An = ASHIFT An BY Dx.L ‡ An = An >>> uimm5 An = An << uimm5
acc40 A_lshift(acc40, short);	An = LSHIFT An BY Dx.L ‡ An = An >> uimm5 An = An << uimm5

Table 1-26. Accumulator Built-In Functions (Cont'd)

Function	Instruction
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);	Dn.lh = An †
fract16 A_madh_FU(acc40);	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);	Dn.lh = An (S2RND) †
short A_madh_ISS2(acc40);	Dn.lh = An (ISS2)
short A_madh_IH(acc40);	Dn.lh = An (IH) †

ⓘ The results of the functions marked with a dagger (†) in Table 1-26 on page 1-248 are affected by the setting of the RND_MOD bit in the ASTAT register. See the *Blackfin Processor Programming Reference* for details.

ⓘ The functions marked with a double dagger (‡) in Table 1-26 on page 1-248 will 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* for details.

Table 1-27. Types Used in Table 1-26 on page 1-248

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.
`fract32`	32-bit signed or unsigned fractional value
`fract16`	16-bit signed or unsigned fractional value
`int`	32-bit signed integer value
`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)
`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 will usually generate an accumulator instruction for each call to an accumulator built-in function, but it will not map `acc40` typed variables to accumulator registers unless optimization is enabled. See the `-O` (enable optimizations) switch on page 1-60.

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;
```

will result 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. Consequently, the two versions of dot product in Listing 1-5 on page 1-252 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 (on page 1-58). This switch will prevent the use of accumulator registers for addition operations but at the expense of reduced performance.

Listing 1-5. Comparison of Two Dot Products

```
#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 Table 1-28 on page 1-256. 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 contains the best solution identified (for example, 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 1-28. 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
`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.

The compiler can be instructed to still generate circular buffer accesses even in such cases, by specifying the `-force-circbuf` switch. (For more information, see "-force-circbuf" on page 1-39.)

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.

```
long circindex(long index, long incr, unsigned long nitems);
```

The operation is equivalent to:

```
index += incr;
if (index < 0)
   index += nitems;
else if (index >= nitems)
   index -= nitems;
```

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);
      }
}
```

Circular Buffer Increment of a Pointer

The following operation performs a circular buffer increment of a pointer.

```
void *circptr(void *ptr, long incr ,
              void * base, unsigned long 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);
      }
}
```

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.

Synchronization

```
void csync_int(void)
```

```
void ssync_int(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.

When it is known that interrupts are disabled at the point a `csync` or `ssync` is required, the `csync_int()` and `ssync_int()` functions may be used instead. These functions issue the `csync` and `ssync` instructions as expected, however the workaround for the 05-00-0312 anomaly (disabling interrupts around the `csync`/`ssync` instruction) will not be applied.

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 will be replaced by the assembly:

```
FLUSH[Preg]; // Preg is loaded with the address __a
```

`__builtin_flush` (data cache line flush) 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 will be replaced by the assembly:

```
FLUSHINV[Preg]; // Preg is loaded with the address __a
```

`__builtin_flushinv` (data cache line flush and invalidate) causes the data cache to perform the same function as `flush` (on page 1-261) 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 will be replaced by the assembly:

```
FLUSHINV[Preg++]; // Preg is loaded with the address __a
```

`__builtin_flushinvmodup` functions exactly the same way as `flushinv` (on page 1-262); 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
```

`__builtin_flushmodup` functions exactly the same way as `flush` (on page 1-261); 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 will be replaced by the assembly:

```
IFLUSH[Preg]; // Preg is loaded with the address __a
```

`__builtin_iflush` (instruction cache flush) 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 will be replaced by the assembly:

```
IFLUSH[Preg++]; // Preg is loaded with the address __a
```

`__builtin_iflushmodup` functions exactly the same way as `iflush` (on page 1-263); 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 will be replaced by the assembly:

```
PREFETCH[Preg]; // Preg is loaded with the address __a
```

`__builtin_prefetch` (data cache prefetch) 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 will be replaced by the assembly:

```
PREFETCH[Preg++]; // Preg is loaded with the address __a
```

`__builtin_prefetchmodup` functions exactly the same way as `prefetch` (on page 1-263); 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 `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 can then 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.

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. These functions generate expanded code to read and write from such memory locations, regardless of whether the access is aligned or not.

```
#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)" on page 1-284 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 (on page 1-52) 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 dereference 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` switch (on page 1-81).

Miscellaneous Built-In Functions

int __builtin_funcsize(const void *func)

The `__builtin_funcsize` built-in function returns the size in bytes of pointer to function func. The result is calculated from the difference between the start and end labels for the function operand. The compiler creates these labels for all C/C++ functions.

The start label is the mangled name of the function. The end label used is a dot ("`.`") followed by the start label followed by "`.end`". For example, for C function foo, these labels are "`_foo:`" and "`._foo.end:`".

When using the `__builtin_funcsize` built-in for assembly functions, the start and end labels need to be correctly defined for it to work.

Example

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

void foo() {
}

void main(void) {
  long size = __builtin_funcsize(foo);
    printf("Function foo is size %ld bytes\n", size);
}
```

The `__builtin_funcsize` built-in does not work for functions defined in different modules than it is used, because end labels are not usually externally visible.

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 C compiler supports pragmas for:

- Arranging alignment of data
- Defining functions that can act as interrupt handlers
- Changing the optimization level, midway through a module
- Changing how an externally visible function is linked
- Providing header file configurations and properties
- Giving additional information about loop usage to improve optimizations

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" on page 1-279
- "Data Alignment Pragmas" on page 1-279
- "Interrupt Handler Pragmas" on page 1-286
- "Loop Optimization Pragmas" on page 1-287
- "General Optimization Pragmas" on page 1-297
- "Fixed-Point Arithmetic Pragmas" on page 1-298
- "Inline Control Pragmas" on page 1-301
- "Linking Control Pragmas" on page 1-303
- "Function Side-Effect Pragmas" on page 1-318
- "Class Conversion Optimization Pragmas" on page 1-330
- "Template Instantiation Pragmas" on page 1-333
- "Header File Control Pragmas" on page 1-335
- "Diagnostic Control Pragmas" on page 1-338
- "Memory Bank Pragmas" on page 1-341
- "Exceptions Tables Pragma" on page 1-347

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 Alignment Pragmas

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.

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.

> 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`.

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 may 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 will only be changed 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 (on page 1-284) or `pad` pragma (on page 1-286) 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     */
```

> 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 (on page 1-280). 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 */
```

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`.

Prior to VisualDSP++ 4.0, this was not the case. The compiler reduced internal alignment, but maintained overall alignment. Since VisualDSP++ 4.0, packed structures may start on different boundaries from unpacked structures. To maintain the overall start alignment, use `#pragma align` (on page 1-279) on the first field of the structure.

#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`*.

Although the `pack` *`alignopt`* pragma emits a warning when a field alignment is reduced, the `pad` *`alignopt`* pragma does not.

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()
```

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; macros are provided by the `sys\exception.h` file. See "Interrupt Handler Support" on page 1-365 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" on page 1-98.)

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.

#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 extra_loop_loads

The `extra_loop_loads` pragma instructs the compiler that the immediately-following loop is allowed to do additional reads past the end of the indicated memory areas, as if the loop were doing an additional iteration, if this allows the compiler to generate faster code. For example,

```
short dotprod_normal(int n, short *x, short *y)
{
   int i;
   short sum = 0;
#pragma no_vectorization
   for (i = 0; i < n; i++)
      sum += x[i] * y[i];
   return sum;
}

short dotprod_with_pragma(int n, short *x, short *y)
{
   int i;
   short sum = 0;
#pragma no_vectorization
#pragma extra_loop_loads
   for (i = 0; i < n; i++)
      sum += x[i] * y[i];
   return sum;
}
```

These examples use the `no_vectorization` pragma to force the compiler to generate simpler versions of the function. Without the `no_vectorization` pragma, the compiler generates vectorized and non-vectorized versions of the loop, which does not invalidate the `extra_loop_loads` pragma, but makes the example more difficult to follow.

In the example, the `dotprod_normal()` function only reads array elements `x[0]..x[n-1]` and `y[0]..y[n-1]`, using the following code:

```
_dotprod_normal:
        P1 = R2 ;
        P2 = R0 ;
        CC = R0 <=  0;
        R0 =    0;
        IF CC JUMP  ._P2L8 ;
        I0 = R1 ;
        P2 +=  -1;
        LSETUP (._P2L5 , ._P2L6-8) LC0 = P2;
        CC =  P2 ==  0;
        MNOP || R0 = W[P1++] (X) || R1.L = W[I0++];
        IF CC JUMP  ._P2L6 ;
.align 8;
._P2L5:
        A0 += R0.L*R1.L (IS) || R0 = W[P1++] (X) ||
              R1.L = W[I0++];
._P2L6:
        A0 += R0.L*R1.L (IS);
        R0 = A0.w;
        R0 = R0.L (X);
._P2L8:
        RTS;
```

The compiler has scheduled the reads from `x[i+1]` and `y[i+1]` in parallel with the addition of `x[i]` and `y[i]`, for best performance. This can only be done for `n`-1 iterations, and so the compiler produces a loop of `n`-1 iterations and does the `n`th addition after the loop terminates. Since `n` is unknown, the compiler must compute `n`-1, and verify that it is not zero before entering the loop.

Compare this with the code generated by the compiler for the function `dotprod_with_pragma()`:

```
_dotprod_with_pragma:
        P1 = R2 ;
        P2 = R0 ;
        CC = R0 <=  0;
        R0 =   0;
        IF CC JUMP  ._P1L8 ;
.align 8;
        I0 = R1 ;
        A0 = 0 || R0 = W[P1++] (X) || NOP;
        R1.L = W[I0++];
        LSETUP (._P1L5 , ._P1L6-8) LC0 = P2;
._P1L5:
        A0 += R0.L*R1.L (IS) || R0 = W[P1++] (X) ||
              R1.L = W[I0++];
._P1L6:
        R0 = A0.w;
        R0 = R0.L (X);
._P1L8:
        RTS;
```

The compiler has generated a loop that has the same instruction in the body of the loop, but here the compiler executes it `n` times, rather than `n`-1 times. This means that the `n`th iteration of the loop will be reading `x[n]` and `y[n]`, which does not happen for `dotprod_normal()`. The values retrieved by these reads are discarded, since they are not needed, but the compiler has gained a benefit because it does not have to compute `n`-1 and determine whether it prevents the loop from executing.

The additional memory reads are only valid if neither `x[]` nor `y[]` are at the end of a valid memory area. If you use the `extra_loop_loads` pragma, you must ensure that the memory ranges within the loop are contiguous with valid memory areas, so that if another iteration's worth of loads is attempted, the loads read from valid addresses.

Note that when the `no_vectorization` pragma is omitted, the compiler will attempt to produce a vectorized loop. The `extra_loop_loads` pragma will not affect the vectorized version, since the compiler will have to conditionally execute a single final iteration anyway, for the cases where the loop count is not an even number.

The `extra_loop_loads` pragma has no effect when:

- The loads are from volatile addresses; such cannot be accessed speculatively
- The loads are from memory banks that cost more than a single cycle to read
- The compiler can determine the number of iterations that the loop will require, either through constant propagation, or through `loop_count` pragmas. In such cases, the compiler does not need to speculatively execute loads.
- The compiler's speed/space ratio prevents it from rotating/pipelining the loop in this manner, because of the increase in code size

See also the `-extra-loop-loads` switch (on page 1-37).

#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_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 may 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, thus providing better software pipelining by the optimizer.

#pragma no_vectorization

The `no_vectorization` pragma turns off all vectorization for the loop on which it is specified.

#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; i<100; 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), 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 which can 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.

#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 will adhere strictly to the ISO/IEC Technical Report 18037 and will convert 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" on page 1-242.

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" on page 1-242.

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 VisualDSP++ 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 VisualDSP++ 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 VisualDSP++ 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).

#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" on page 1-159 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" on page 1-143.

```
#pragma inline
int func5(int a, int b) {   /* can be inlined */
   return a / b;
}
```

#pragma never_inline

The `never_inline` pragma may be applied to a function definition to indicate to the compiler that 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" on page 1-159 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;
}
```

Linking Control Pragmas

Linking control pragmas (`linkage_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 (on page 1-355).

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
*/
}
```

#pragma core

When building a project that targets multiple processors or multiple cores on a processor, a link stage may produce executable files 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 analyses 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 ad infinitum. 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 may be 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 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 multiprocessor 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 startup) 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 analyses correctly.

ⓘ 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 `#pragma core` 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.

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" on page 1-98.

#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 VisualDSP++ tools will 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 runtime.
- Members/fields within `structs/unions/classes`
- Type declarations

For more information on IPA, see "Interprocedural Analysis" on page 1-98.

#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 x=5;
#pragma default_section(DATA, "NEW_DATA3")
int x;
```

In this case, `x` is placed in `NEW_DATA2` because the definition of `x` 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 Table 1-29.

Table 1-29. 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 uninitialized 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" on page 1-419.
`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 Table 1-30.

Table 1-30. 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
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;
```

ⓘ 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" on page 1-427.

#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" will be used.

> 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” on page 1-471 for more information on using attributes.

#pragma symbolic_ref

The `symbolic_ref` pragma may 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(){}
```

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” on page 1-98.

Function Side-Effect Pragmas

Function side-effect pragmas (`alloc`, `pure`, `const`, `inline`, `misra_func`, `noreturn`, `regs_clobbered`, `overlay`, 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 `a` and `b` cannot overlap `out`.

```
#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 (on page 1-321). 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 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 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.

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;
}
```

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.

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.

When the compiler detects an illegal string, a warning is issued and the default volatile set is used instead. (See "Scratch Registers" on page 1-433.)

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.

Register `P1` is used by the linker to expand `CALL` instructions, so it may be modified at the call site regardless of whether the `regs_clobbered` pragma says it is clobbered. Therefore, the compiler never keeps `P1` live across a call. However, the compiler accepts the pragma when compiling a function in case the user wants 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.

User-Reserved Registers

User-reserved registers, indicated via the `-reserve` switch (on page 1-71), 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 registers 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 convention 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 for 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;
}
```

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.

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" on page 1-466 for more information on how the compiler handles templates.

The instantiation pragmas take the name of an instance as a parameter, as shown in Table 1-31.

Table 1-31. 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.

> 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 (`hdrstop`, `no_implicit_inclusion`, `no_pch`, `once`, and `system_header`) help the compiler to handle header files.

#pragma hdrstop

The `hdrstop` pragma is used with the `-pch` (precompiled header) switch (on page 1-66). The `-pch` switch instructs the compiler to look for a precompiled header (`.pch` file), and, if it cannot find one, to generate a file for use on a later compilation. The `.pch` file contains a snapshot of all the code preceding the header stop point.

By default, the header stop point is the first non-preprocessing token in the primary source file. The #pragma hdrstop can be used to set the point earlier in the source file.

In the following example, the default header stop point is the start of the declaration i.

```
#include "standard_defs.h"
#include "common_data.h"
#include "frequently_changing_data.h"

int i;
```

This might not be a good choice, as "frequently_changing_data.h" might change frequently, causing the .pch file to be regenerated often, and, therefore, losing the benefit of precompiled headers. The hdrstop pragma can be used to move the header stop to a more appropriate place.

In the following example, the precompiled header file would not include the contents of frequently_changing_data.h, as it is included after the hdrstop pragma, and so the precompiled header file would not need to be regenerated each time frequently_changing_data.h was modified.

```
#include "standard_defs.h"
#include "common_data.h"
#pragma hdrstop
#include "frequently_changing_data.h"

int i;
```

#pragma no_implicit_inclusion

With the -c++ switch (on page 1-26), 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 no_pch

The no_pch pragma overrides the -pch (precompiled headers) switch (on page 1-66) for a particular source file. It directs the compiler not to look for a .pch file and not to generate one for the specified source file.

#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
```

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 VisualDSP++. The VisualDSP++ 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 three 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 Table 1-32.

Table 1-32. Keywords for ACTION Qualifier

Keyword	Action
`suppress`	Suppresses all instances of the diagnostic
`remark`	Changes the severity of the diagnostic to a remark
`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{error|remark|suppress|warn}" on page 1-79.

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`.

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)`
  This pragma can be used to inhibit all subsequent warnings and remarks (equivalent to the `-w` switch option).

- `#pragma diag(remarks)`
  This pragma can be used to enable all subsequent remarks and warnings (equivalent to the `-Wremarks` switch option)

- `#pragma diag(warnings)`
  This pragma can be used to restore the default behavior when neither `-w` or `-Wremarks` is specified, which is to display warnings but inhibit remarks.

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)`
  This pragma may be used to store the current state of the severity of all diagnostic error messages.

- `#pragma diag(pop)`
  This 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`, `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.

Directives to modify specific diagnostics are singular (for example, “`error`”), and the directives to modify classes of diagnostics are plural (for example, “`errors`”).

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.

Note that memory banks are different from sections:

- Section is a "hard" placement, 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" placement, 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).

#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 a bank called "`__code`". 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.

In the following example, the `add_slowly()` function is placed into the "`slowmem`" bank, which may have different performance characteristics from the "`__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. Without this pragma, the compiler assumes that non-local data should use the bank "`__data`" for behavioral characteristics.

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 memory bank "__data", 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 memory bank "__stack".

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 memory bank "`__stack`", 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;
}
#include <sys/exception.h>
#pragma stack_bank(sysstack)
EX_INTERRUPT_HANDLER(count_ticks)
{
      extern int ticks;
      ticks++;
}
```

#pragma bank_memory_kind(*bankname, kind*)

The `bank_memory_kind` pragma informs the compiler of what *`kind`* of memory the memory bank *`bankname`* is. The compiler allows the following kinds of memory:

- Internal – The memory bank is high-speed in-core memory
- L2 – The memory bank is on-chip, but not in-core
- External – The memory bank is external to the processor

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*)

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 schedule sufficient code between the initiation of the read and the use of its results, to prevent unnecessary stalls.

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*)

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 schedule sufficient code between the initiation of the write and a subsequent read or write to the same location, to prevent unnecessary stalls.

In the following example, the compiler knows that each write through `ptr` to the “`output`” memory bank takes six cycles to complete.

```
void write_buf(int n, const char *buf)
{
      volatile bank("output") char *ptr = REG_ADDR;
      while (n--)
          *ptr = *buf++;
}
#pragma bank_write_cycles(output, 6)
```

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition. This is shown in the preceding example.

#pragma bank_optimal_width(*bankname, width*)

The `bank_optimal_width` pragma informs the compiler that *`width`* is the optimal number of bits to transfer to/from memory bank *`bankname`* in a single cycle. This can be used to indicate to the compiler that some memories can benefit from vectorization and similar strategies more than others. The *`width`* parameter must be 8, 16, 24, or 32.

In the following example, the compiler knows that the instructions for the generated function would be best fetched in multiples of 16 bits, and so can select instructions accordingly.

```
void memcpy_simple(char *dst, const char *src, size_t n)
{
      while (n--)
          *dst++ = *src++;
}
#pragma bank_optimal_width(__code, 16)
```

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition. This is shown in the preceding example.

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 (on page 1-35) 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 3.4 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/onlined-ocs/gcc-3.4.6/gcc/index.html#toc_C-Extensions`

The GCC compatibility extensions are only available in C dialect mode. They are not accepted in C++ dialect mode.

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;
}
```

Statement expressions are not supported in C++ mode. 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;
```

> The `typeof` keyword is not supported in C++ mode.
> 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.

GCC Generalized lvalues

A cast is an `lvalue` (may appear on the left-hand side of an assignment) if its operand is an `lvalue`. This is an extension to C, provided for compatibility with GCC. It is not allowed in C++ mode.

A comma operator is an `lvalue` if its right operand is an `lvalue`. This is an extension to C, provided for compatibility with GCC. It is a standard feature of C++.

A conditional operator is an `lvalue` if its last two operands are `lvalues` of the same type. This is an extension to C, provided for compatibility with GCC. It is a standard feature of C++.

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.

The following example calls `lookup()` once, and substitutes the string "`-`" if it returns NULL. This is an extension to C, provided for compatibility with GCC. It is not allowed in C++ mode.

```
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 trace(file,line,msg ...) \
        logmsg(file,line, ## msg);
```

can be used with differing numbers of arguments,

```
trace("a.c", 22, "Got here!\n");

trace("b.c", 99, "i = %d\n", i);

trace("c.c", 72, "x = %f, y = %f\n", x, y);
```

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.

ⓘ The variable argument macro syntax comes from GCC. The compiler supports both GCC and C99 variable argument macro formats in C89, C99, and C++ modes. (“Variable Argument Macros” on page 1-164).

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. The extension is not compatible with many dialects of C, including ANSI/ISO C89 and C99. However, it is useful in `asm` statements, which are intrinsically non-portable.

This extension may be disabled via the `-no-multiline` switch on page 1-57.

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.

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 member's types.

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 `...` (three periods).

For example,

```
case 200 ... 300:
```

Escape Character Constant

The escape character "`\e`" may be used in character and string literals. It maps to the ASCII Escape code, 27.

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 *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" on page 1-304.)

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 would use 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. All attributes accepted by GCC on `ix86` are accepted. Only attributes with corresponding pragmas (see “Pragmas” on page 1-277) will be used by the compiler; all other attributes are ignored.

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
```

Blackfin Processor-Specific Functionality

This section provides information about functionality that is specific to Blackfin processors.

This section describes:

- "Startup Code Overview" on page 1-357
- "Support for argv/argc" on page 1-358
- "Profiling With Instrumented Code" on page 1-359
- "Controlling System Heap Size and Placement" on page 1-364
- "Interrupt Handler Support" on page 1-365
- "Caching and Memory Protection" on page 1-373

Startup Code Overview

Startup code, which is invoked when the processor starts running, initializes a default environment before calling `main()`. The VisualDSP++ **Project Wizard** can be used to generate startup code based on specified options.

If you select not to add a generated CRT in the **Project Wizard**, your application will normally link using a pre-built default CRT from the <install_path>\Blackfin\lib folder in the VisualDSP++ installation. The source for these default CRT objects can be found in <install_path>\Blackfin\lib\src\libc\crt\basiccrt.s.

If you decide not to use a generated file but instead to customize the startup code, copy the `basiccrt.s` source into your project and make the desired customizations. If you are using a default `.ldf` file, you must define the `USER_CRT` linker macro. Refer to "C/C++ Run-Time Header and Startup Code" on page 1-410 for more information.

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";
  ```

- To use command-line arguments as part of profile-guided optimization (PGO), it is necessary to define `__argv_string` within a memory section called `MEM_ARGV`. Therefore, define a memory section called `MEM_ARGV` in your `.ldf` file and include the definition of `__argv_string` in it if you are using PGO. The default `.ldf` files do this for you if macro `IDDE_ARGS` is defined at link-time. They do this using a RESOLVE statement to map __argv_string to the start of MEM_ARGV. For this to succeed, it can be necessary for the definition of __argv_string to be preceded by #pragma symbolic_ref.

Profiling With Instrumented Code

The profiling facilities determine how many times each function is called and how many cycles are used while the function is active. The information is gathered by an additional library linked into the executable file. The profiling routine is invoked by additional function calls at the start and end of each function. The compiler inserts these extra calls when profiling is enabled.

The compiler profiling facilities are different from linear profiling and statistical profiling features.

The compiler profiling facilities are designed for single-core and single-threaded systems. The compiler driver issues warning cc3106 if either the -multicore switch (on page 1-50) or -threads switch (on page 1-76) is used together with the -p[1|2] switch (on page 1-65).

Generating Instrumented Code

The -p[1|2] switch (on page 1-65) turns on the compiler's profiling facility when converting C/C++ source into assembly code. The compiler cannot instrument assembly files or files that have already been compiled to object files.

- The -p1 option causes the generated application to write accumulated profile data to file mon.out.
- The -p2 option causes the generated application to write accumulated profile data to standard output.

- The -p option causes the generated application to write accumulated profile data to both standard output and the mon.out file.

When created, the mon.out file will reside in the same folder as the application is run.

Running the Executable

The executable may produce two forms of output. The first (generated by -p and -p2) is a dump of data to standard output once the program completes. This output lists the approximate address of each profiled function, how many times the function was invoked, and the inclusive and exclusive cycle counts.

- Exclusive cycle counts include only the cycles spent processing the function.
- Inclusive cycle counts also include the sum total of cycle counts in any function invoked from this specified function.
- The cycle counts generated are the total cycles spent in all invocations of the specified function within the program.

The second form of output is a file in the current directory called mon.out (-p and -p1). The mon.out is a binary file that contains a copy of the data written to standard output. There is no way to change the file name used.

For example, in the following program, assume that apple() takes 10 cycles per call and banana() takes 20 cycles per call, of which 10 are accounted for by its call to apple().

```
int apples, bananas;
void apple(void) {
    apples++;        // 10 cycles
}
```

```
void banana(void) {
    bananas++;        // 10 cycles
    apple();          // 10 cycles
}                     // 20 cycles total

int main(void) {
    apple();          // 10 cycles
    apple();          // 10 cycles
    banana();         // 20 cycles
    return 0;         // 40 inclusive cycles total
}                     // + exclusive cycles for main itself
```

When run, the program 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 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, and `banana()` once). Between them, `apple()` and `banana()` use up to 40 cycles, which appear in the `main()`'s inclusive cycles. The `main()`'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()`.

Example of `stdout` profiling output:

```
version=2 nrecs=3 Profiler cycles=5818
Addr:ffa096c4  ExecCount:   1 ExCyc:   10 IncCyc:   10
Addr:ffa0967c  ExecCount:   3 ExCyc:   30 IncCyc:   30
Addr:ffa0969e  ExecCount:   1 ExCyc:   10 IncCyc:   20
```

Post-Processing the mon.out File

The `profblkfn` program processes the contents of the `mon.out` file. It reads both the `mon.out` file and the `.dxe` file that produced it. It displays:

- `Function Name` – The name of the function being profiled
- `ExecCount` – The number of times the function is called
- `Fn Only` – The total number of cycles spent processing this function; that is, the "exclusive cycle count"
- `Fn+nested` – The total number of cycles spent processing this function and any functions it calls; that is, the "inclusive cycle count"

The `profblkfn` program is invoked as:

```
profblkfn prog.dxe
```

Specify the `.dxe` file only. The `mon.out` file must be present in the current directory and must be produced by the named `.dxe` file.

Example of `profblkfn` output:

```
Function Name  ExecCount    Fn Only  Fn+nested
_main             1         10        50
_apple            3         30        30
_banana           1         10        20
```

where:

`ExecCount` is the number of times the function is executed. `Fn Only` is the total cycle count for all executions of the function ignoring any function calls made within that function (for example, each call to `_banana` is 10 cycles plus the call to `_apple`; so the value of `Fn Only` for `_banana` is 10 cycles per call to `_banana`). `Fn+nested` is the total cycle count (`Fn Only`) of the function, plus the individual cycle counts of any other functions that are called.

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 it is exhausted), the profiling runtime issues an error ("*Profiler Resource Error: heap allocation failed so profiling cannot be completed*") and stops storing information. The application will continue to execute but may fail if the application also uses the system heap. The profiling data available when this happens will be incomplete and will probably not be very useful. To avoid this problem, increase the size of the system heap until the error is no longer seen when running. See "Controlling System Heap Size and Placement" on page 1-364 for details.

Computing Cycle Counts

When profiling is enabled, the compiler instruments the generated code by inserting calls to a profiling library at the start 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, whose overhead is 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 of error. This error margin is more significant for functions with a small number of instructions than for functions with a large number of instructions.

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.

`.ldf` files created by the Project Wizard can be controlled using selections on the **LDF Settings : System Heap** page of the **Project Options** dialog box. 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 will be 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

```
// 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 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.
- `USE_L1DATA_HEAP` – (default) Places the heap in L1 data bank 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.

See "Using Multiple Heaps" on page 1-423 for more information.

Interrupt Handler Support

The Blackfin C/C++ compiler provides support for interrupts and other events used by the Blackfin processor architecture (Table 1-33).

The Blackfin system has several different classes of events, not all of which are supported by the `ccblkfn` compiler. Handlers for these events are called *interrupt service routines* (ISRs).

Table 1-33. System Events

Event	Priority	Supported
Emulation	Highest	No
Reset		Yes
NMI		Yes
Exception		Yes
Interrupt	Lowest	Yes

Resets are supported by treating them like general-purpose interrupts for code-generation purposes. This means that the C/C++ compiler supports interrupt, exception, and NMI events.

The compiler provides facilities for defining an `ISR` function, registering it as an event handler, and for obtaining the saved processor context.

Defining an ISR

To define a function as an ISR, the `sys/exception.h` header file must be included and the function must be declared and defined using macros defined within this header file. There is a macro for each of the three kinds of events the compiler supports:

```
EX_INTERRUPT_HANDLER
EX_EXCEPTION_HANDLER
EX_NMI_HANDLER
```

By default, ISRs generated by the compiler are not re-entrant; they disable the interrupt system on entry, and re-enable it on exit. You may also define ISRs for interrupts that are re-entrant, and which re-enable the interrupt system soon after entering the ISR.

A different macro is used to specify a re-entrant interrupt handler:

```
EX_REENTRANT_HANDLER
```

For example, the following code declares and defines my_isr() as a handler for interrupt-type events (for example, the routine returns using an RTI instruction).

```
#include <sys/exception.h>
static volatile int number_of_interrupts;

EX_INTERRUPT_HANDLER(my_isr)
{
   number_of_interrupts++;
}
```

The macro used for defining the ISR is also suitable for declaring it as a prototype:

```
EX_INTERRUPT_HANDLER(my_isr);
```

The EX_INTERRUPT_HANDLER() macro uses a generic pragma, #pragma interrupt, to indicate that the function is an interrupt handler. This generic pragma does not indicate which interrupt the function handles. The -workaround isr-imask-check switch selection (on page 1-81) for hardware anomaly 05-00-0071 on the ADSP-BF535 processor requires explicit information on the level of interrupt being handled, so that the interrupt can be re-raised if the interrupt is taken while a CLI instruction is being committed.

Such an ISR is defined as:

```
EX_HANDLER_PROTO(interrupt_level_6, my_handler){
}
```

Eleven level-specific pragmas, 5 through 15, correspond to the Blackfin event table entries for interrupts.

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.

While thread-safe variants of the C/C++ run-time libraries exist, many functions are not interrupt-safe as they access global data structures. It is therefore recommended that ISRs do not make library function calls, as unexpected behavior may result if the interrupt occurs during a call to such a function. An alternative approach is to disable interrupts before the application makes run-time library calls. This may be disadvantageous for time-critical applications as interrupts may be disabled for a long period of time. The DSP run-time library functions do not modify global data structures and are therefore interrupt-safe.

To define a static ISR, place the "`static`" qualifier within the appropriate macro's brackets – but not before the macro itself; for example:

```
#include <sys/exception.h>
EX_REENTRANT_HANDLER(static Sport1_TX_ISR)
{
// ISR code
}
```

Registering an ISR

ISRs, once defined, can be registered in the event vector table (EVT) using the `register_handler_ex()` or `register_handler()` functions, both of which also update the `IMASK` register so that the interrupt can take effect. Only the `register_handler_ex()` function will be discussed here, as it is an extended version of the `register_handler()` function. Refer to "register_handler_ex" on page 3-270 for more information.

The `register_handler_ex()` function takes three parameters, defining the event, the ISR, and specifying whether the interrupt should be enabled, disabled, or left in its current state. It also returns the previously registered

ISR (if any). The event is specified using the `interrupt_kind` enumeration from `exception.h`.

```
typedef enum {
  ik_emulation, ik_reset, ik_nmi, ik_exception,
  ik_global_int_enable, ik_hardware_err, ik_timer,ik_ivg7,
  ik_ivg8, ik_ivg9, ik_ivg10, ik_ivg11, ik_ivg12, ik_ivg13,
  ik_ivg14, ik_ivg15
} interrupt_kind;
ex_handler_fn register_handler_ex(interrupt_kind kind,
           ex_handler_fn fn, int enable);
```

Two special values of *`fn`* can be passed to `register_handler_ex()` in place of real ISRs:

- `EX_INT_IGNORE`
  Leaves the currently-installed handler in place, but disables the interrupt (subject to the `enable` parameter)
- `EX_INT_DEFAULT`
  Clears the event vector table entry for this event, so no handler is installed, and disables the interrupt (subject to the `enable` parameter)

The `enable` parameter may have one of the following values:

- `EX_INT_KEEP_IMASK`
  Causes the event handler to be installed without changing the "enabled" status of the event. If the event was previously enabled, it remains so. If the event was previously disabled, it remains so. This value has no effect if *`fn`* is `EX_INT_DISABLE` or `EX_INT_IGNORE`.
- `EX_INT_DISABLE`
  Causes the event to be disabled before installing the new handler; the event will be disabled on return from `register_handler_ex()`. This value has no effect if *`fn`* is `EX_INT_DISABLE` or `EX_INT_IGNORE`.

- `EX_INT_ENABLE`
  Causes the event to be enabled after installing the new handler; the event will be enabled on return from `register_handler_ex()`. This value has no effect if *`fn`* is `EX_INT_DISABLE` or `EX_INT_IGNORE`.

- `EX_INT_ALWAYS_ENABLE`
  Causes the event to be enabled after installing the new handler; the event will be enabled on return from `register_handler_ex()`. This value takes effect even if *`fn`* is `EX_INT_DISABLE` or `EX_INT_IGNORE`.

ISRs and ANSI C Signal Handlers

ISRs provide similar functionality to ANSI C signal handlers, and their behavior is related. An ISR is a function that can be registered directly in the processor's event vector table (EVT). The `ISR` function saves its own context, as required. In contrast, an ANSI C signal handler is a normal C function that has been registered as a handler; when an event occurs, some other dispatcher must save the processor context before invoking the signal handler.

ISRs and signal handlers are not interchangeable. A signal handler cannot act as an ISR, because it does not save or restore the context, nor does it terminate with the correct return instruction. An ISR cannot act as a signal handler, because it terminates the event directly rather than returning to the dispatcher.

When a signal handler is installed, a default ISR is also installed in the EVT which invokes the signal handler when the event occurs. When the `raise()` function is used to invoke a signal handler explicitly, `raise()` actually generates the corresponding event (if possible). This causes the ISR to invoke the signal handler.

You may choose to install normal C functions as signal handlers or register ISRs directly, but do not do both for a given event.

ANSI C signals are registered using `signal()` or using the Analog Devices extension `interrupt()`, unlike ISRs, which are registered using `register_handler_ex()` or `register_handler()`.

Saved Processor Context

When generating code for an ISR, the compiler creates a prologue that saves the processor context on the supervisor stack. This context is accessible to the ISR. The `exception.h` file defines a structure, `interrupt_info`, that contains fields for all the information that defines the kind of event that occurred.

To save an event's context (in the handler), the `get_interrupt_info` function can be called. The prototype for `get_interrupt_info()` is:

```
void get_interrupt_info(
         interrupt_kind int_kind, interrupt_info *int_info);
```

An example use of `get_interrupt_info()` would be to save interrupt information for later use as shown in the example below:

```
#include <sys/exception.h>
static interrupt_info last_int_info;

EX_INTERRUPT_HANDLER(ivg7_fielder)
{
   get_interrupt_info(ik_ivg7, &last_int_info);
   // handle the interrupt
}
```

The `get_interrupt_info()` function does not provide facilities to save register values.

Fetching Event Details

The following function fetches the information about the event that occurred:

```
void get_interrupt_info(interrupt_kind, interrupt_info *)
```

The sort of data retrieved includes the value of `SEQSTAT` and addresses that caused exceptions. Note that at present, the function must be told which kind of event it is investigating.

The structure contains:

```
interrupt_kind kind;
int value;
void *pc;
void *addr;
unsigned status;
```

These fields are set as:

- **Exceptions**
  The `pc` field is set to the value of `RETX`, and `value` is set to the value of `SEQSTAT`.
  For exceptions that involve address faults, the `addr` and `status` fields are set to the values of the memory-mapped registers (MMRs) for `DATA_FAULT_ADDR` and `DATA_FAULT_STATUS` or for `CODE_FAULT_ADDR` and `CODE_FAULT_STATUS`, as appropriate.

- **Hardware Errors**
  The `pc` field is set to the value of `RETI`, and `value` is set to the value of `SEQSTAT`.

- **NMI Events**
  The `pc` field is set to the value of `RETN`.

- **All Other Events**
  The `pc` field is set to the value of `RETI`.

Caching and Memory Protection

Blackfin processors support the caching of external memory or L2 SRAM (where available) into L1 SRAM, for both instruction and data memory. Caching can eliminate much of the performance penalty of using external memory with minimal effort on the application developer's part.

This section describes:

The Blackfin processor caches are configurable devices. Instruction and data caches can be enabled together or separately, and the memory spaces they cache are configured separately. The cache configuration is defined through the memory protection hardware, using tables that define cacheability protection lookaside buffers (CPLBs). These CPLBs define the start addresses, sizes, and attributes of areas of memory for which memory accesses are permitted (including whether the area of memory is to be cached).

Refer to the appropriate Blackfin processor's *Hardware Reference* for details.

The Blackfin run-time library provides support for cache configuration by providing routines that can be used to initialize and maintain the CPLBs from a configuration table.

Both the **Project Wizard**-generated C/C++ run-time (CRT) headers and default pre-compiled CRT objects use these library routines. The default configuration does not enable CPLBs. The support routines are designed such that they can easily be incorporated into users' systems, and so that the configuration can be turned on or off via a debugger, without having to re-link the application. (See "C/C++ Run-Time Header and Startup Code" on page 1-410 for more information.)

___cplb_ctrl Control Variable

CPLB support is controlled through a global integer variable, `___cplb_ctrl`. Its C name has two leading underscores, and its assembler name has three leading underscores. The value of this variable determines whether the startup code enables the CPLB system. By default, the variable has the value 0 (zero), indicating that CPLBs are not enabled.

The variable's value is a bitmask, based on the macros defined in the `<cplb.h>` header. The macros are:

- `CPLB_ENABLE_ICPLBS`
  Turns on instruction CPLBs
- `CPLB_ENABLE_ICACHE`
  Turns on instruction caching into L1 Instruction memory

- `CPLB_ENABLE_DCPLBS`
  Turns on data CPLBs
- `CPLB_ENABLE_DCACHE`
  Turns on data caching into L1 Data A memory
- `CPLB_ENABLE_DCACHE2`
  Turns on data caching into L1 Data B memory
- `CPLB_SET_DCBS`
  Sets the data cache bank select bit in the `DMEM_CONTROL` register. This specifies which bit of a memory address determines the data cache bank (A or B) used to cache the location. Depending on the placement of data within the application memory space, one setting or the other ensures more data is cached at runtime. This bit has no effect unless both `CPLB_ENABLE_DCACHE` and `CPLB_ENABLE_DCACHE2` bits are also set. Refer to the processor's *Hardware Reference* for further details.

These macros are OR'd together to produce the value for `___cplb_ctrl`.

Note that:

- If `CPLB_ENABLE_DCACHE2` is set, `CPLB_ENABLE_DCACHE` must also be set.
- If any of the three cache bits are set, the corresponding `CPLB_ENABLE_ICPLBS` or `CPLB_ENABLE_DCPLBS` bit must also be set.
- `___cplb_ctrl` must be placed in a locked CPLB.

There is a default definition of `___cplb_ctrl` in the C run-time library, which defaults to disabling CPLBs and caching. This default definition is overridden by any definition in the CRT startup code generated by the

Project Wizard, or alternatively by providing your own definition within your application. For example,

```
#include <stdio.h>
#include <cplb.h>
#pragma section("cplb_data")
int __cplb_ctrl =         // C syntax with two underscores
   CPLB_ENABLE_ICPLBS |
   CPLB_ENABLE_ICACHE;
int main(void) {
   printf("Hello world\n");
   return 0;
}
```

The new definition enables CPLBs and turns on instruction caching; data caching is not enabled.

CPLB Installation

When `___cplb_ctrl` indicates that CPLBs are to be enabled, the startup code calls the routine `_cplb_init`. This routine sets up instruction and data CPLBs from a table, and enables the memory protection hardware.

There are sixteen CPLBs for each instruction and data space. On a simple system, this is sufficient, and `_cplb_init` installs all available CPLBs from its configuration table into the active table. On more complex systems, there may need to be more CPLBs than can be active at once. In such systems, a time may come when the application attempts to access memory that is not covered by one of the active CPLBs. This raises a CPLB miss exception.

For these occasions, the library includes a CPLB management routine, `_cplb_mgr`. This routine should be called from an exception handler that has determined that a CPLB miss has occurred (either a data miss or an instruction miss). The `_cplb_mgr` routine identifies the inactive CPLB that needs to be installed to resolve the access, and replaces one of the active CPLBs with this one.

If CPLBs are to be enabled, the default startup code installs a default exception handler called `_cplb_hdr`; this does nothing except test for CPLB miss exceptions, which it delegates to `_cplb_mgr`. It is expected that users have their own exception handlers that deal with additional events.

If data CPLBs are enabled, it is necessary to ensure that `__cplb_ctrl` is mapped to data that is covered by a locked CPLB as it is loaded in the default exception handler (`cplb_hdr`) prior to calling `cplb_mgr` to handle CPLB exceptions. This can be done by using a `#pragma` section to define `___cplb_ctrl` in a section that is mapped to a memory range that is covered by a locked CPLB. The default and generated `.ldf` files provide sections that can be used for this purpose.

It is not possible to recover from a CPLB miss that occurs when handling a prior miss exception. To avoid this, ensure that the code and data used when handling a CPLB miss is covered by an active CPLB. The CPLB management code is placed into a section called `cplb_code`. The data used is the stack to save and restore registers and the variable `___cplb_ctrl`.

It is necessary to ensure that the CPLBs for these are:

- Flagged as being “locked”, so they are not replaced by inactive CPLBs during misses
- Flagged as “dirty” if the caching mode is set to write-back mode

The `cplb_data` section is used to contain the CPLB configuration tables. It is not necessary to have a locked CPLB covering this section because the CPLB management code disables CPLBs before accessing the data these tables contain.

When enabling CPLBs, `_cplb_init` checks that the CPLB entries are valid. If an issue is identified, control will jump to an infinite loop around a label describing the problem. These labels are described in Table 1-34.

Table 1-34. CPLB Issues

Label	Error
`cplb_address_is_misaligned_for_cplb_size`	Alignment of CPLB does not correspond to CPLB size. Each CPLB must have a minimum alignment equal to the size of the CPLB.
`too_many_locked_data_cplbs`	More than 16 locked data CPLBs are present. Only 16 data CPLBs are available, so additional data CPLBs cannot become active.
`too_many_locked_instruction_cplbs`	More than 16 locked instruction CPLBs are present. Only 16 instruction CPLBS are available, so additional instruction CPLBs cannot become active.

Cache Configurations

Although CPLBs may be used for their protection capabilities, most often they are used to enable caching. The `___cplb_ctrl` variable is the means by which the application directs the run-time library to install CPLBs for caching.

The library defines the following configurations, although not all configurations may be available on all Blackfin processors:

- No cache
- L1 SRAM Instruction as cache
- L1 SRAM Data A as cache
- L1 SRAM Data A and B as cache

- L1 SRAM Instruction and Data A as cache
- L1 SRAM Instruction, Data A and Data B as cache

Note that if any cache is enabled, the corresponding data or instruction CPLBs must also be enabled. Furthermore, if you are using the default `.ldf` files, you must also tell the linker that the cache is enabled; this is discussed in more detail in "Default Cache Configuration" and "Default .ldf Files and Cache" on page 1-385.

If any cache is enabled, the respective caches are set up during `_cplb_init`, using the CPLB configuration tables. On ADSP-BF535 processors, if cache is enabled, the current cache contents are invalidated using the functions described in "Cache Invalidation" on page 1-383. With other Blackfin processors, the cache is automatically invalidated at power-up.

Default Cache Configuration

Although the default value for `___cplb_ctrl` is that no cache or CPLBs are enabled, the default system contains CPLB configuration tables that permit caching. The default configuration tables differ for the parts available.

The default configuration tables are defined in files called `cplbtab`*n*`.s` in `VisualDSP/Blackfin/lib/src/libc/crt`, where *n* is the part number.

Table 1-35 lists the default CPLB configuration files.

Table 1-35. Default CPLB Configuration Files

Blackfin Processor	Configuration file
ADSP-BF504	`cplbtab504.s`
ADSP-BF504F	`cplbtab504f.s`
ADSP-BF506F	`cplbtab506f.s`
ADSP-BF512	`cplbtab512.s`
ADSP-BF514	`cplbtab514.s`

Table 1-35. Default CPLB Configuration Files (Cont'd)

Blackfin Processor	Configuration file
ADSP-BF516	cplbtab516.s
ADSP-BF518	cplbtab518.s
ADSP-BF522	cplbtab522.s
ADSP-BF523	cplbtab523.s
ADSP-BF524	cplbtab524.s
ADSP-BF525	cplbtab525.s
ADSP-BF526	cplbtab526.s
ADSP-BF527	cplbtab527.s
ADSP-BF531	cplbtab531.s
ADSP-BF532	cplbtab532.s
ADSP-BF533	cplbtab533.s
ADSP-BF534	cplbtab534.s
ADSP-BF535	cplbtab535.s
ADSP-BF536	cplbtab536.s
ADSP-BF537	cplbtab537.s
ADSP-BF538	cplbtab538.s
ADSP-BF539	cplbtab539.s
ADSP-BF542	cplbtab542.s
ADSP-BF542M	cplbtab542M.s
ADSP-BF544	cplbtab544.s
ADSP-BF544M	cplbtab544M.s
ADSP-BF548	cplbtab548.s
ADSP-BF548M	cplbtab548M.s
ADSP-BF549	cplbtab549.s
ADSP-BF549M	cplbtab549M.s

Table 1-35. Default CPLB Configuration Files (Cont'd)

Blackfin Processor	Configuration file
ADSP-BF561 (Core A)	`cplbtab561a.s`
ADSP-BF561 (Core B)	`cplbtab561b.s`

If memory protection or caching has been selected through the VisualDSP++ Project Wizard, you are allowed to generate a customizable CPLB table. For more information, refer to the description of the "VisualDSP++ Project Wizard" available from VisualDSP++ Help.

Each file defines two tables:

1. `icplbs_table[]` – Instruction CPLBs
2. `dcplbs_table[]` – Data CPLBs

The table's structure is defined by `cplbtab.h`, specifying the start address of each area of memory, and the controlling attributes for that area. The definitions of the macros that are used to define these attributes are contained in the `defblackfin.h` standard include file and are documented in the appropriate *Hardware Reference* manual.

The default tables include areas of memory for L1 SRAM, internal L2 (where present), external asynchronous and SDRAM memory, and other memory spaces. The external areas are configured to be cacheable using write-through mode by default. If no cache is enabled and CPLBs are enabled, the run-time library masks off the cacheable flags on the CPLBs before making them active.

The tables are defined by OR'ing a combination of the macros defined in `cplb.h` and the core-specific header files (`def_LPBlackfin.h` or `defblackfin.h`). The macros in `cplb.h` define bitmasks that specify common CPLB configurations.

A brief description of these macros follows:

- `CPLB_I_PAGE_MGMT`
  Default instruction CPLB configuration for memory page covering page management code in `cplb_code` section. The CPLB is locked (so it cannot be evicted), and valid.
- `CPLB_DEF_CACHE`
  Default data cache configuration – memory page is cached in write-through mode
- `CPLB_DEF_CACHE_WT`
  Same as `CPLB_DEF_CACHE`
- `CPLB_DEF_CACHE_WB`
  Default data cache configuration – memory page is cached in write-back mode
- `CPLB_ALL_ACCESS`
  Memory protection properties – specifies all accesses are allowed to this page
- `CPLB_DNOCACHE`
  All accesses are allowed, CPLB is valid, but page is not cached
- `CPLB_DDOCACHE`
  Same as `CPLB_DNOCACHE`, but page is cached
- `CPLB_DDOCACHE_WT`
  Same as `CPLB_DNOCACHE`, but page cached in write-through mode
- `CPLB_DDOCACHE_WB`
  Same as `CPLB_DNOCACHE`, but page cached in write-back mode

- `CPLB_INOCACHE`
  Instruction memory read-only access, CPLB is valid, page is not cached

- `CPLB_IDOCACHE`
  Same as `CPLB_INOCACHE`, but page is cached

None of the above macros specify a page size, so they should be OR'd with `PAGE_SIZE_1KB`, `PAGE_SIZE_4KB`, `PAGE_SIZE_1MB`, or `PAGE_SIZE_4MB` (defined in core-specific header) as appropriate.

Changing Cache Configuration

The value of `___cplb_ctrl` may be changed in several ways:

- The **Project Wizard** can be used to generate CRT startup code that includes a definition of the `___cplb_ctrl` variable, based on the selected cache configuration.

- It may be defined as a new global variable with an initialization value. This definition supersedes the definition in the library. The example in "___cplb_ctrl Control Variable" on page 1-374 uses this approach.

- The linked-in version of the variable may be altered in a debugger, after loading the application but before running it, so that the startup code sees a different value.

Cache Invalidation

The `cache_invalidate` routine may be used to invalidate the processor's instruction and/or data caches. It is defined as:

```
#include <cplbtab.h>

void cache_invalidate(int cachemask);
```

Its parameter is a bitmask, indicating which caches should be cleared.

Table 1-36. Bitmasks and Caches to be Cleared

Bit set	Cache invalidated
`CPLB_ENABLE_ICACHE`	Instruction cache
`CPLB_ENABLE_DCACHE`	Data cache A
`CPLB_ENABLE_DCACHE2`	Data cache B

The `cache_invalidate` routine uses several supporting routines:

```
#include <cplbtab.h>
void icache_invalidate(void);
void dcache_invalidate(int a_or_b);
void dcache_invalidate_both(void);
```

The `icache_invalidate` routine clears the instruction cache.

The `dcache_invalidate` routine clears a single data cache, determined by the `a_or_b` parameter:

- `CPLB_INVALIDATE_A` invalidates data cache A
- `CPLB_INVALIDATE_B` invalidates data cache B

The `dcache_invalidate_both` routine clears data cache A and data cache B. On ADSP-BF535 processors, this is done by calling `dcache_invalidate` for each cache. On other Blackfin processors, it toggles control bits in the `DMEM_CONTROL` register, which invalidates the contents of both data caches in a single operation.

The `dcache_invalidate` and `dcache_invalidate_both` routines do not flush any modified cache entries to memory first, if any memory pages are cached in write-back mode. To flush such data prior to invalidation, use the functions described in "Cache Flushing" on page 1-389.

Default .ldf Files and Cache

The default `.ldf` files supplied with VisualDSP++ are designed to support caching with minimal effort.

The default `.ldf` files have three basic configurations:

1. No external SDRAM and no caching. All code and data are placed into internal SRAM. This is the default configuration.
2. External SDRAM and no caching. Code and data are placed into both internal SRAM and external SDRAM. Code and data are placed into internal SRAM where possible. This configuration is enabled by passing the `-MDUSE_SDRAM` flag to the linker at link-time.
3. External SDRAM and caching enabled. This will require one or more of the LDF caching macros to be defined when linking.

Configuration 1 is most efficient but is not suitable for larger applications that will not fit into internal memory.

Configuration 2 allows larger applications to occupy external memory but they will incur significant performance overheads when running code or accessing data that is mapped to external memory.

Configuration 3 is an efficient configuration for larger applications (than would fit in L1) as it allows larger applications to use external memory while minimizing the performance overhead by using the cache hardware. As mentioned previously, this configuration requires the definition of one or more macros when using the default `.ldf` file. These macros are used to ensure that the `.ldf` file does not map code or data to memory that will be configured as cache. These macros are normally defined using the linker's `-MD` switch and must match the cache configuration defined by `___cplb_ctrl`.

Table 1-37 lists these macros.

Table 1-37. Macros for Caching

Macro	Function
USE_INSTRUCTION_CACHE	Indicates that L1 instruction SRAM is reserved for use as cache, for example, when ___cplb_ctrl is defined with CPLB_ENABLE_ICACHE.
USE_DATA_A_CACHE	Indicates that L1 data bank A is reserved for use as cache, for example, when ___cplb_ctrl is defined with CPLB_ENABLE_DCACHE.
USE_DATA_B_CACHE	Indicates that L1 data bank B is reserved for use as cache, for example, when ___cplb_ctrl is defined with CPLB_ENABLE_DCACHE2.
USE_CACHE	Indicates that caching is being enabled. It must be defined when one or more of USE_INSTRUCTION_CACHE, USE_DATA_A_CACHE, or USE_DATA_B_CACHE is defined. If none of USE_INSTRUCTION_CACHE, USE_DATA_A_CACHE, and USE_DATA_B_CACHE are defined when USE_CACHE is the default, the .ldf file works as if USE_INSTRUCTION_CACHE, USE_DATA_A_CACHE, and USE_DATA_B_CACHE were all defined.

If cache is not enabled, the macros listed above should not be used or the memory they refer to will be unused.

Therefore, if the .ldf file believes cache is to be used, but ___cplb_ctrl specifies otherwise, resources are wasted, but the application still functions. In fact, this is the case if only code or data caches are requested by ___cplb_ctrl, but not both.

The last entry of Table 1-37 shows a configuration that must be avoided since mapping code/data into L1 SRAM, which is then configured as cache, leads to corrupt code/data. This scenario can also be difficult to debug, so the run-time library provides a mechanism for protecting against this case, such as:

- The default `.ldf` files define global "guard" symbols, setting their addresses to be 0 or 1, according to the LDF caching macros (`USE_INSTRUCTION_CACHE`, `USE_DATA_A_CACHE`, and `USE_DATA_B_CACHE`) that are defined at link-time. If objects are mapped into a cache area during linking, the guard symbol is set to 0 (indicating this cache area is not available); otherwise, it is set to 1 (indicating that the cache area is available).
- When `_cplb_init` is enabling CPLBs and cache, the run-time library tests the guard symbols. If a cache has been requested via `___cplb_ctrl`, but the corresponding guard symbol indicates that the cache area has already been allocated during link-time, the library signals an error. It does so by jumping to an infinite loop around labels with names that describe the problem.

These are defined as follows:

```
l1_code_cache_enabled_when_l1_used_for_code:
   JUMP 0;
l1_data_a_cache_enabled_when_used_for_data:
   JUMP 0;
l1_data_b_cache_enabled_when_used_for_data:
   JUMP 0;
```

The guard symbols have the following names:

`___l1_code_cache`

`___l1_data_cache_a`

`___l1_data_cache_b`

CPLB Replacement and Cache Modes

As previously noted, no more than 16 CPLBs may be active concurrently for each instruction space or data space. Large applications may need to address more memory than this, and may eventually access a memory location not covered by the currently-active CPLBs. At this point, a CPLB "miss exception" occurs, and the application's exception handler must select one of the active CPLBs for removal to make way for a new CPLB that covers the address being accessed. This victimization and replacement process is handled by the `_cplb_mgr` routine within the run-time library. The process varies, depending on which cache modes are active.

Blackfin processors support two variants of caching: write-through mode and write-back mode.

- In write-through mode, writes to cached memory are written to both the cache and the memory location. Consequently, write-through mode primarily provides performance gains for memory reads. The memory location is kept up-to-date.
- In write-back mode, writes to cached memory are only written to the cache. They are not written to the memory location until the cache line is victimized (by an access to another memory location) or flushed (through programmatic means).

The cache mode (write-through, write-back, or off) is specified on a per-CPLB basis, so one page may be cached in write-through mode, another in write-back mode, and a third not cached at all.

By default, write-back pages are "clean", in that they do not have the `DIRTY` flag set. When a write occurs to a clean write-back page, a protection violation exception is raised to indicate that the page is being written to. The `_cplb_mgr` routine flags the page's CPLB as `DIRTY`, and allows the write to continue. This time, it succeeds. If the `DIRTY` flag is set when the CPLB is first installed, no exception will be generated on first write.

This DIRTY flag can be used to identify which pages may contain data not yet propagated back to memory; if the cache needs to propagate data back to memory so that it can evict the data and cache another address, the DIRTY flag will not be cleared.

Because write-through pages always update the memory location with the new cached value, write-through pages need not be marked as DIRTY. Consequently, write-through pages do not trigger an exception on first write to the page.

The victimization process chooses victim CPLBs in the following order of preference:

1. Unused (for example, invalid) CPLBs
2. Unlocked CPLBs

Note that only unlocked CPLBs are selected as victims. Locked CPLBs are never selected. In particular, it is necessary to ensure that the CPLB management routines reside in pages that are covered by locked CPLBs to prevent the CPLB management routines from evicting themselves.

To assist in this, the CPLB management routines reside in the cplb_code section. This section must be explicitly mapped to memory that is covered by a locked and valid CPLB. It is also necessary to ensure that the data stack and cache control variable ___cplb_ctrl is always valid in the same way.

Cache Flushing

If desired, write-back data can be flushed back to memory using the flush_data_cache routine. The routine searches all active pages for valid, modified pages that are cached in write-back mode, and flushes their contents back to memory. This time-consuming process is dependent on the size of the modified data page.

Table 1-38. Flushing Costs per Page Size

Page Size	Approximate cost of flushing
1K	400 instructions
4K	1500 instructions
1M	6000 instructions
4M	6000 instructions

The costs in Table 1-38 are approximate, because they only take into account the number of instructions executed, and do not include the costs of data transfers from cache to external memory. The actual cost is greatly influenced by the amount of modified data residing in the caches.

If it is necessary to ensure that smaller areas of memory are flushed to memory, the `flush_data_buffer` routine may be used:

```
#include <cplbtab.h>

void flush_data_buffer(void *start, void *end, int invalidate);
```

This routine flushes back to memory any changes in the data cache that apply to the address range specified by the `start` and `end` parameters. If the `invalidate` parameter is non-zero, the routine also invalidates the data cache for the address range, so that the next access to the range will require a re-fetch from memory.

Using the _cplb_mgr Routine

The `_cplb_mgr` routine is intended to be invoked by the application's exception handler. The source for `_cplb_mgr` can be found within your VisualDSP++ installation in the `Blackfin/lib/src/libc/crt/cplbmgr.s` file. A minimal exception handler, `_cplb_hdr`, is installed by the default startup code, and its source is in `Blackfin/lib/src/libc/crt/cplbhdr.s`.

Typically, the exception handler delegates CPLB misses and protection violations by calling `_cplb_mgr` and handles all other exceptions itself. The `_cplb_mgr` routine is defined as (in C nomenclature):

```
int cplb_mgr(int code, int cplb_ctrl);
```

where `code` indicates the kind of exception raised (Table 1-39).

Table 1-39. Exception Mask Codes

Code Value	Meaning
0	Instruction CPLB miss
1	Data CPLB miss
2	Protection violation (assumed to be first-write to write-back data page)

The routine accepts the current value of `___cplb_ctrl` as the second parameter.

There are several error codes that `_cplb_mgr` can return, defined in `<cplb.h>` as shown in Table 1-40.

Table 1-40. CPLB Return Codes

Return Code	Meaning
`CPLB_RELOADED`	Successfully updated CPLB table
`CPLB_NO_UNLOCKED`	All CPLBs are locked; thus, they cannot be evicted. This indicates that the CPLBs in the configuration table are badly configured, as this should never occur.
`CPLB_NO_ADDR_MATCH`	The address being accessed, that triggered the exception, is not covered by any of the CPLBs in the configuration table. The application is presumably misbehaving.
`CPLB_PROT_VIOL`	The address being accessed, that triggered the exception, is not a first-write to a clean write-back data page, and so presumably is a genuine violation of the page's protection attributes. The application is misbehaving.

If _cplb_mgr returns an error indicator, the exception handler must decide how to handle the error. The default exception handler installed by the startup code delegates each of these failure conditions—plus one other—to handler functions in the run-time library.

These functions are:

```
void _unknown_exception_occurred(void);
void _cplb_miss_all_locked(void);
void _cplb_miss_without_replacement(void);
void _cplb_protection_violation(void);
```

These functions are stubs that can be replaced for comprehensive error handling. They enter an infinite loop with verbose labels, indicating the kind of error that has occurred. For assistance when debugging, automatic breakpoints are placed on these functions.

The _cplb_mgr routine modifies the following registers:

R0-R3, P0-P2, I0-I2, ASTAT, LC0, LC1, LB0, LB1, LT0, and LT1

It is therefore necessary that an exception handler, calling _cplb_mgr, saves these registers before calling _cplb_mgr and restores them before returning from the exception.

Caching and Asynchronous Change

Care must be taken when using the cache in systems with asynchronous change. There are two levels of asynchronous data change:

- Data that may change beyond the scope of the current thread, but within the scope of the system. This includes variables that may be updated by other threads in the system (if using a multi-threaded architecture). This kind of data must be marked volatile, so that the compiler knows not to store local copies in registers (but may be located in cached memory), since all threads access the data through the cache.

- Data that may change beyond the scope of the cache as well as beyond the scope of the current thread. This includes memory-mapped registers (which cannot be cached) and data in memory that is updated by external means, such as DMA transfers or host/target file I/O. Such data must be marked as volatile, so that the compiler knows not to keep copies in registers. This data may not be placed in cached memory since the cache does not see the change and provides date copies to the application. Alternatively, the cache copy must be invalidated before accessing memory, in case it has been updated.

Migrating .ldf Files From Previous VisualDSP++ Installations

The `.ldf` files which have been used in VisualDSP++ 4.5 projects require updating before they can be used in VisualDSP++ 5.0.

For customized `.ldf` files, you must make the changes manually.

For `.ldf` files generated by the Project Wizard, these changes can be applied automatically, as follows:

1. Open the project using VisualDSP++ 5.0. The VisualDSP++ IDDE will ask for confirmation before upgrading the project to VisualDSP++5.0. Click "**Yes**".
2. In **Project Options, LDF Settings**, change one of the settings, and click on **OK**. The Project Wizard will regenerate your `.ldf` file.
3. In **Project Options, LDF Settings**, change the setting back to its original value and click on **OK**. The Project Wizard will regenerate your `.ldf` file again. Your `.ldf` file will now be ready for use.

The changes are described in:

- "C++ Support Tables (ctor, gdt)" on page 1-394
- "Dual-Core Single-Application Per Core Shared Data" on page 1-395
- "C++ Run-Time Libraries Rationalization" on page 1-396
- "Multi-Threaded Libraries" on page 1-397
- "Fixed-Point I/O Support" on page 1-399

C++ Support Tables (ctor, gdt)

This change is required.

Linker changes in VisualDSP++ 5.0 make it possible for non-contiguous placement of highly-aligned data. This means that order of mapping in output memory sections is not necessarily maintained. This will result in linker warning `li2040`, which can be avoided by using the `FORCE_CONTIGUITY` directive when contiguous placement is required, and `NO_FORCE_CONTIGUITY` otherwise.

The C++ static constructor mechanism (`ctor/ctorl`) and exceptions handling support (`.gdt/.gdtl`) use table inputs terminated using the sections ending in "`l`". This requires contiguous placement of these sections, so use of `FORCE_CONTIGUITY` is recommended.

For example, replace:

```
L1_data_b {
   INPUT_SECTION_ALIGN(4)
   INPUT_SECTIONS( $OBJECTS(L1_data_b) $LIBRARIES(L1_data_b))
   INPUT_SECTIONS( $OBJECTS(ctor) $LIBRARIES(ctor))
   INPUT_SECTIONS( $OBJECTS(ctorl) $LIBRARIES(ctorl))
   INPUT_SECTIONS( $OBJECTS(.gdt) $LIBRARIES(.gdt))
   INPUT_SECTIONS( $OBJECTS(.gdtl) $LIBRARIES(.gdtl))
```

```
    // ...
} >MEM_L1_DATA_B
```

with:

```
/* one-to-one mapping first */
L1_data_b {
    INPUT_SECTION_ALIGN(4)
    INPUT_SECTIONS( $OBJECTS(L1_data_b) $LIBRARIES(L1_data_b))
} >MEM_L1_DATA_B

L1_data_b_tables {
    INPUT_SECTION_ALIGN(4)
    FORCE_CONTIGUITY
    INPUT_SECTIONS( $OBJECTS(ctor)  $LIBRARIES(ctor))
    INPUT_SECTIONS( $OBJECTS(ctorl) $LIBRARIES(ctorl))
    INPUT_SECTIONS( $OBJECTS(.gdt)  $LIBRARIES(.gdt))
    INPUT_SECTIONS( $OBJECTS(.gdtl) $LIBRARIES(.gdtl))
} >MEM_L1_DATA_B

L1_data_b {
    INPUT_SECTION_ALIGN(4)
    // ...
} >MEM_L1_DATA_B
```

For more information, see “Constructors and Destructors of Global Class Instances” on page 1-419.

Dual-Core Single-Application Per Core Shared Data

> This change is required for dual-core profiles that use the single-application/dual-core approach.

When linking the core B `.dxe` file of a single application per core multi-core configuration (see “One Application Per Core” on page A-7), it is necessary to ensure that shared data is resolved by linking against the core A .dxe file rather than a core-specific definition. If the linker sees a RESOLVE directive for a symbol linked locally and separately in the core, it will issue warning li2143.

There is a particular case that can cause this to happen in the default .ldf files which has been avoided in VisualDSP++ 5.0. The change was to delete the use of mc_data561.doj by removing the following lines:

```
#if defined(__ADI_MULTICORE) && defined(COREA)
   RT_OBJ_NAME(mc_data561),    /* multi-core shared data */
#endif
```

and modifying the use of libmc*.dlb to use a linker attribute filter to ensure that core B does not link a local instance of shared library data. This is done by modifying:

```
#if defined(__ADI_MULTICORE)
   RT_LIB_NAME(mc561),    /* multi-core library */
#endif
```

to:

```
#if defined(COREB)
   RT_LIB_NAME(mc561) {!sharing("MustShare")},
      /* multi-core shared data */
#else
   RT_LIB_NAME(mc561),
      /* multi-core library */
#endif
```

C++ Run-Time Libraries Rationalization

This change is optional.

In previous versions of VisualDSP++, it was necessary to link against libcpp*.dlb, libcpprt*.dlb, and libx*.dlb when C++ exceptions support was required. In VisualDSP++ 5.0, it is only necessary to link against the libcpp*.dlb library. Therefore, it is possible to simplify your .ldf file by removing references to the libx*.dlb and libcpprt*.dlb libraries.

Multi-Threaded Libraries

This change is optional.

In VisualDSP++ 5.0 Update 8 and earlier, the -threads switch did not link against thread-safe libraries unless the application used VDK. As of VisualDSP++ 5.0 Update 9, non-VDK .ldf files will also use the thread-safe libraries when the -threads switch is specified.

The changes in the default .ldf files are not trivial. They are controlled by the presence of the _ADI_THREADS link-time macro, which the compiler driver automatically defines when -threads is specified. There are two types of change:

1. New macros RT_LIB_NAME_MT(n) and RT_LIB_NAME_EH_MT(n) are defined. These will specify the libraries used depending on whether the -eh switch is active. The definitions on the macros depend on _ADI_THREADS: if _ADI_THREADS is defined, the macros name the thread-safe libraries, otherwise they name the non-thread-safe libraries.
2. Libraries which are delivered in thread-safe and non-thread-safe flavors are identifies using these two new macros.

As an example of the first case, consider the file ADSP-BF548.ldf. In VisualDSP++ 5.0 Update 8, this file contains the following definitions:

```
#  define RT_LIB_NAME(x) lib ## x ## y.dlb
#  define RT_OBJ_NAME(x) x ## y.doj
#  if defined(__ADI_LIBEH__)
#    define RT_LIB_NAME_EH(x) lib ## x ## yx.dlb
#  else /* __ADI_LIBEH__ */
#    define RT_LIB_NAME_EH(x) lib ## x ## y.dlb
#  endif
```

In VisualDSP++ 5.0 Update 9, these definitions have been augmented with choices dependent on the presence of _ADI_THREADS:

```
#  define RT_LIB_NAME(n) lib ## n ## y.dlb
#  define RT_OBJ_NAME(n) n ## y.doj
#  if defined(_ADI_THREADS)
#    define RT_LIB_NAME_MT(n) lib ## n ## mty.dlb
#    if defined(__ADI_LIBEH__)
#      define RT_LIB_NAME_EH_MT(n) lib ## n ## mtyx.dlb
#    else /* __ADI_LIBEH__ */
#      define RT_LIB_NAME_EH_MT(n) lib ## n ## mty.dlb
#    endif
#  else /* _ADI_THREADS */
#    define RT_LIB_NAME_MT(n) lib ## n ## y.dlb
#    if defined(__ADI_LIBEH__)
#      define RT_LIB_NAME_EH_MT(n) lib ## n ## yx.dlb
#    else /* __ADI_LIBEH__ */
#      define RT_LIB_NAME_EH_MT(n) lib ## n ## y.dlb
#    endif
#  endif /* _ADI_THREADS */
```

Consider the same file again, for an example of the second set of changes. In VisualDSP++ 5.0 Update 8, the file contains the following library list (comments removed for clarity):

```
$LIBRARIES =
   RT_LIB_NAME(small532),
...Other libraries elided...
#if defined(USE_FILEIO) || defined(USE_PROFGUIDE)
   RT_LIB_NAME(rt_fileio532),
#else
   RT_LIB_NAME(rt532),#endif
   RT_LIB_NAME(event532),
   RT_LIB_NAME_EH(cpp532),
#if defined(IEEEFP)
   RT_LIB_NAME(sftflt532),
#endif
```

```
...Other libraries elided...
RT_LIB_NAME(profile532)
    ;
```

In VisualDSP++ 5.0 Update 9, the same list now uses the new macros (once again, comments removed for clarity):

```
$LIBRARIES =
    RT_LIB_NAME_MT(small532),
...Other libraries elided..
#if defined(USE_FILEIO) || defined(USE_PROFGUIDE)
    RT_LIB_NAME_MT(rt_fileio532),
#else
    RT_LIB_NAME_MT(rt532),
#endif
    RT_LIB_NAME_MT(event532),
    RT_LIB_NAME_EH_MT(cpp532),
#if defined(IEEEFP)
    RT_LIB_NAME(sftflt532),
#endif
...Other libraries elided...
    RT_LIB_NAME(profile532)
    ;
```

Notice that not all the libraries in the list employ the new macros—not all libraries require thread-safety.

Fixed-Point I/O Support

This change is only required if your application requires formatted-I/O support for fixed-point types.

As of VisualDSP++ 5.0 Update 9, fixed-point types are natively supported by the compiler, and formatted-I/O support is optionally available, when the `_ADI_FX_LIBIO` macro is defined at link time. This is achieved by linking against a different I/O library when the macro is defined.

For example, ADSP-BF548.ldf in VisualDSP++ 5.0 Update 8 contains the following definitions (comments removed for clarity):

```
#if defined(_DINKUM_IO)
   RT_LIB_NAME(c532),
   RT_LIB_NAME(io532),
#else
   RT_LIB_NAME(io532),
   RT_LIB_NAME(c532),
#endif
```

In VisualDSP++ 5.0 Update 9, the definitions have been augmented by the _ADI_FX_LIBIO macro, which is automatically defined by the compiler driver when the -fixed-point-io switch is specified at link time (once again, comments removed for clarity):

```
#if defined(_DINKUM_IO)
   RT_LIB_NAME_MT(c532),
   RT_LIB_NAME_MT(io532),
#else
#if defined(_ADI_FX_LIBIO)
   RT_LIB_NAME_MT(iofx532),
#else
   RT_LIB_NAME_MT(io532),
#endif
   RT_LIB_NAME_MT(c532),
#endif
```

Notice that the definitions also use the new macros for selecting the thread-safe libraries if required; see "Multi-Threaded Libraries" on page 1-397.

C/C++ Preprocessor Features

Several features of the C/C++ preprocessor are used by VisualDSP++ 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:

`//`	end of line (C++ style) commands
`#warning`	directive

For more information about these extensions, see "Preprocessor-Generated Warnings" on page 1-357 and "C++ Style Comments" on page 1-173. For ways to write macros, refer to "Writing Preprocessor Macros" on page 1-405.

This section contains:

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.

For the list of predefined assertions, see "-A" on page 1-27.

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.

Table 1-41 describes the predefined compiler macros.

Table 1-41. Predefined Compiler Macros

Macro	Function
`_ADI_FX_LIBIO`	Defined as 1 when compiling with the `-fixed-point-io` switch.
`_ADI_COMPILER`	Defined as 1.
`__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. **Note**: This does not include the ADSP-BF535 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.
`__ADSPBLACKFIN__`	Always defined as 1.

Table 1-41. Predefined Compiler Macros (Cont'd)

Macro	Function
`__ADSPLPBLACKFIN__`	Defined as 1 when the target processor (set using the `-proc` switch) is one of low-power core parts. These include ADSP-BF504, ADSP-BF504F, ADSP-BF506F, ADSP-BF512, ADSP-BF514, ADSP-BF516, ADSP-BF518, ADSP-BF522, ADSP-BF523, ADSP-BF524, ADSP-BF525, ADSP-BF526, ADSP-BF527, ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF534, ADSP-BF536, ADSP-BF537, ADSP-BF538, ADSP-BF539, ADSP-BF542, ADSP-BF547, ADSP-BF548, ADSP-BF549, ADSP-BF561, or ADSP-BF592-A processors.
`__ADSPBF506F_FAMILY__`	Defined as 1 when the target processor (set using the `-proc` switch) is the ADSP-BF504, ADSP-BF504F, or ADSP-BF506F processor.
`__ADSPBF518_FAMILY__`	Equivalent to `__ADSPBF51x__`.
`__ADSPBF526_FAMILY__`	Equivalent to `__ADSPBF52xLP__`.
`__ADSPBF527_FAMILY__`	Equivalent to `__ADSPBF52x__`.
`__ADSPBF533_FAMILY__`	Equivalent to `__ADSPBF53x__`.
`__ADSPBF535_FAMILY__`	Defined as 1 when the target processor (set using the `-proc` switch) is the ADSP-BF535.
`__ADSPBF537_FAMILY__`	Equivalent to `__ADSPBF53x__`.
`__ADSPBF538_FAMILY__`	Equivalent to `__ADSPBF53x__`.
`__ADSPBF548_FAMILY__`	Equivalent to `__ADSPBF54x__`.
`__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.
`__ADSPBF592_FAMILY__`	Defined as 1 when the target processor (set using the `-proc` switch) is the ADSP-BF592-A processor.
`__ANALOG_EXTENSIONS__`	Defined as 1. If MISRA compliance checking is enabled, this macro will not be defined.
`__cplusplus`	Defined as `199711L` when you compile in C++ mode.
`__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 1-41. Predefined Compiler Macros (Cont'd)

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, the macro is not defined.
`__ECC__`	Always defined as 1.
`__EDG__`	Always defined as 1. This definition signifies that an Edison Design Group 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 on page 1-35).
`__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).
`_INSTRUMENTED_PROFILING`	Defined as 1 when instrumented profiling is enabled (using the `-p` switches on page 1-65).
`_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).
`_MISRA_RULES`	Defined as 1 when compiling in MISRA-C mode.
`__NO_BUILTIN`	Defined as 1 when you compile with the `-no-builtin` command-line switch (on page 1-53).
`__NUM_CORES__`	Defined to be the number of cores in the currently-selected target processor. For example, when compiling for the ADSP-BF533 processor, `__NUM_CORES__` is defined as 1, whereas when compiling for the ADSP-BF561 processor, `__NUM_CORES__` is defined as 2.
`__RTTI`	Defined as 1 when C++ run-time type information is enabled (using the `-rtti` switch on page 1-90).
`__SIGNED_CHARS__`	Defined as 1, unless you compile with the `-unsigned-char` command-line switch (on page 1-78).

Table 1-41. Predefined Compiler Macros (Cont'd)

Macro	Function
`__STDC__`	Always defined as 1.
`__STDC_VERSION__`	Defined as 199409L when compiling in C89 mode, and as 199901L when compiling in C99 mode.
`__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`.
`__VISUALDSPVERSION__`	The preprocessor defines this macro to be an eight-digit hexadecimal representation of the VisualDSP++ release, in the form `0xMMmmuurr`, where: – `MM` is the major release number – `mm` is the minor release number – `uu` is the update number – `rr` is "00", and is reserved for future use For example, VisualDSP++5.0 Update 1 would be `0x05000100`.
`__WORKAROUNDS_ENABLED`	Defines this macro to be 1 if any hardware workarounds are implemented by the compiler. This macro is set if the `-si-revision` switch (on page 1-74) has a value other than "`none`" or if any specific workaround is selected by means of the `-workaround` switch (on page 1-81).

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.

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.

Analog Devices recommends that assembly programmers maintain stack conventions.

Figure 1-2 provides an overview of the run-time environment issues that must be considered when writing assembly routines that link with C/C++ routines including the "C/C++ Run-Time Header and Startup Code" on page 1-410. The run-time environment issues include the following items.

- Memory usage conventions

  "Using Memory Sections" on page 1-422

  "Using Multiple Heaps" on page 1-423

  "Using Data Storage Formats" on page 1-443

- Register usage conventions

  "Dedicated Registers" on page 1-432

  "Call-Preserved Registers" on page 1-433

  "Scratch Registers" on page 1-433

  "Stack Registers" on page 1-435

- Program control conventions

"Managing the Stack" on page 1-435

"Transferring Function Arguments and Return Value" on page 1-439

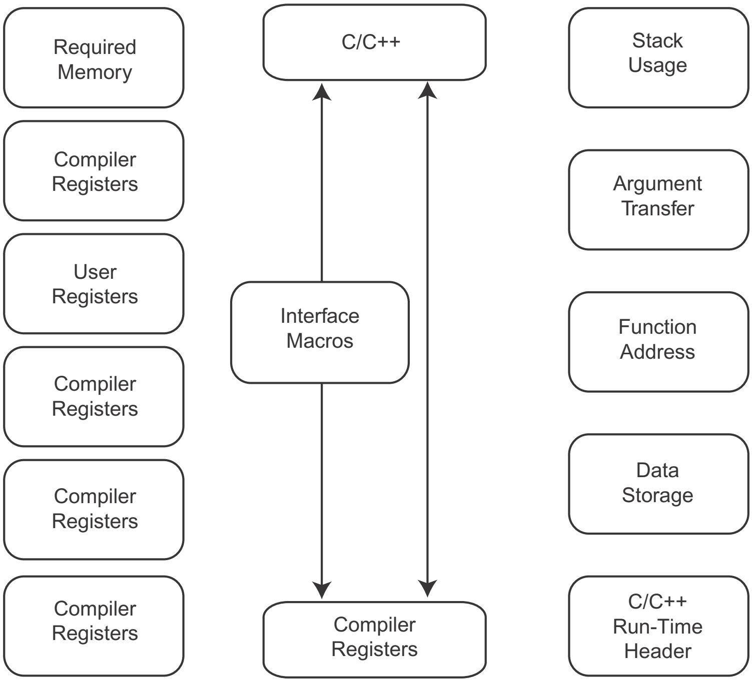


Figure 1-2. Assembly Language Interfacing Overview

C/C++ Run-Time Header and Startup Code

The C/C++ run-time (CRT) header is code that is executed after the processor jumps to the start address on reset. The CRT header sets the machine into a known state and calls `_main`. CRT code can be included in a project in one of the following ways:

- The **Project Wizard** can be used to automatically generate a customized CRT in a project. Refer to VisualDSP++ Help for details.
- The macro `USER_CRT` can be defined at link-time to specify a custom user-defined CRT object is to be included in the project build.
- Default CRT objects are provided for all platforms in the run-time libraries, and are linked against for all C/C++ projects if the link-time macro `USER_CRT` is not defined.

This section contains:

- "CRT Header Overview"
- "CRT Description" on page 1-412

CRT Header Overview

The CRT ensures that when execution enters `_main`, the processor's state obeys the C application binary interface (ABI), and that global data declared by the application have been initialized as required by the C/C++ standards. It arranges things so that `_main` appears to be "just another function" invoked by the normal function invocation procedure.

Not all applications require the same configuration. For example, C++ constructors are invoked only for applications that contain C++ code. The list of optional configuration items is long enough that determining whether to invoke each one in turn at runtime would be overly costly.

For this reason, the **Project Wizard** allows a CRT to be generated which includes the minimal amount of code necessary given the user-selected options. Additionally, the pre-built CRTs are supplied in several different configurations, which can be specified at link-time via LDF macros.

The CRT header is used for projects that use C, C++, and VDK. Assembly language projects do not provide a default run-time header; you must provide your own.

The source assembly file for the pre-compiled CRTs is located under the VisualDSP++ installation directory, in the file `basiccrt.s`, in the directory `Blackfin/lib/src/libc/crt`. Each of the pre-built CRT objects are built from this default CRT source. The different configurations are produced by the definition of various macros.

The list of operations performed by the CRT (startup code) can include (not necessarily in the following order):

- Setting registers to known/required values
- Disabling hardware loops
- Disabling circular buffers
- Setting up default event handlers and enabling interrupts
- Initializing the stack pointer and frame pointer
- Enabling the cycle counter
- Configuring the memory ports used by the two DAGs
- Copying data from the flash memory to RAM
- Initializing device drivers
- Setting up memory protection and caches
- Changing processor interrupt priority
- Initializing profiling support

- Invoking C++ constructors
- Invoking `_main`, with supplied parameters
- Invoking `_exit` on termination

What the CRT Does Not Do

The CRT does not initialize actual memory hardware. The initialization of the external SDRAM is left to the boot loader because it is possible (and even likely) that the CRT itself will need to be moved into external memory before being executed.

CRT Description

The following sections describe the main operations that may be performed by the CRT, dependent on the selected **Project Wizard** options, or which of the pre-built CRTs is included in the build.

Declarations

The CRT begins with preprocessor directives that "include" the appropriate platform-definition header and set up a few constants:

- `IVBl` and `IVBh` give the address of the event vector table
- `UNASSIGNED_VAL` is a bit pattern that indicates that the register/memory location has not yet been written to by the application. See "Mark Registers" on page 1-417 and "Terminate Stack Frame Chain" on page 1-418.

- INTERRUPT_BITS is the default interrupt mask. By default, it enables the lowest-priority interrupt, IVG15. This default mask can also be overridden at runtime by your own version of __install_default_handlers; see "Event Vector Table" on page 1-413 for details.

- For some platforms, SYSCFG_VALUE is the initialization value for the system configuration register (SYSCFG).

Start and Register Settings

The CRT declares its first code label as start. This required label is referenced by .ldf files, which explicitly resolve this label to the processor's reset address.

First, the CRT disables facilities that could be enabled on start-up, due to their random power-up states, as follows:

- SYSCFG is set to SYSCFG_VALUE, according to anomaly 05-00-0109 for ADSP-BF531, ADSP-BF532, ADSP-BF533, and ADSP-BF561 processors.

- Hardware loops are disabled to prevent jump-back-to-loop-start behavior, should the "loop bottom" register correspond to the start of an instruction.

- Circular buffer lengths are set to zero. The CRT makes use of the Iregs and calls functions that may use them. Furthermore, the C/C++ ABI requires that circular buffers are disabled on entry to (and exit from) compiled functions, so the circular buffers must be disabled before invoking _main.

Event Vector Table

The reset vector (fixed) and emulation events (not touched by the C ABI), are not defined by the CRT. The processor's lowest-priority event, IVG15, is set to point to supervisor_mode, a label that appears later in the CRT and is used to facilitate the switch to supervisor mode. The remaining entries of the event vector table are loaded with the address of the `__unknown_exception_occurred` dummy event handler, which results in defined behavior to aid debugging.

Additionally, if caching or memory protection is enabled (either selected via the **Project Wizard**, or configured by the user-defined value of the `___cplb_ctrl` variable), an exception handler is required to process possible events raised by the memory system. Therefore, the default handler, `___cplb_hdr`, is installed into the exception entry of the event vector table.

For details on `___cplb_ctrl`, refer to "Caching and Memory Protection" on page 1-373.

You may install additional handlers; for your convenience, the CRT calls a function to do this. The function, `__install_default_handlers`, is an empty stub, which you may replace with your own function that installs additional or alternative handlers, before the CRT enables events. The function's C prototype is:

```
short _install_default_handlers(short mask);
```

The CRT passes the default enable mask, (`INTERRUPT_BITS`), as a parameter, and considers the return value to be an updated enable mask. If you install additional handlers, you must return an updated enable mask to reflect this.

> See the *VisualDSP++ Kernel (VDK) User's Guide* for details on how to configure ISRs for applications that use VDK.

Stack Pointer and Frame Pointer

The stack pointer (`SP`) is set to point to the top of the stack, as defined in the `.ldf` file by the symbol `ldf_stack_end`. Specifically, the stack pointer is set to point just past the top of the stack. Because stack pushes are pre-decrement operations, the first push moves the stack pointer so that it refers to the actual stack top.

The user stack pointer (USP) and frame pointer (FP) are set to point to the same address.

Twelve bytes are then claimed from the stack. This is because the C ABI requires callers to allocate stack space for the parameters of callees, and that all functions require at least twelve bytes of stack space for registers R0-R2. Therefore, the CRT claims these twelve bytes as the incoming parameters for functions called before invoking _main.

Cycle Counter

The CRT enables the cycle counter, so that the CYCLES and CYCLES2 registers are updated. This is not necessary for general program operation, but is desirable for measuring performance.

DAG Port Selection

For ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF534, ADSP-BF536, ADSP-BF537, ADSP-BF538, ADSP-BF539, and ADSP-BF561 processors, the CRT configures the DAGs to use different ports for accessing memory. This reduces stalls when the DAGs issue memory accesses in parallel.

Memory Initialization

Memory initialization is a two-stage process:

1. At link-time, the Memory Initializer utility processes the .dxe file to generate a table of code and data memory areas that must be initialized during booting.

2. At runtime, when the application starts, the run-time library function _mi_initialize processes the table to copy the code and data from the flash device to volatile memory.

If the application has not been processed by the Memory Initializer, or if the Memory Initializer did not find any code or data that required such movement, the _mi_initialize function returns immediately. If the

"**Enable run-time memory initialization**" option is selected in the **Project Wizard**, the generated CRT includes a call to `_mi_initialize`. The default CRT source always includes the call.

The CRT does not enable external memory. The configuration of physical memory hardware is the responsibility of the boot loader and must be complete before the CRT is invoked.

Device Initialization

The process of initializing device drivers that support `stdio` involves:

1. Initializing the internal file tables
2. Invoking the initialization routine for each device driver registered at build-time
3. Associating `stdin`, `stdout`, and `stderr` with the default device driver

By default, this process occurs automatically when a device is first accessed. For information on the device drivers supported by stdio, refer to "Extending I/O Support to New Devices" on page 3-44.

If the **C/C++ I/O and I/O device support** option on the **Run-Time Initialization** page of the **Project Wizard** is selected (which is the default), explicit device initialization is included in the generated CRT. Support for the device drivers for stdio may be disabled under the **Project Wizard** by de-selecting the option.

CPLB Initialization

When cacheability protection lookaside buffers (CPLBs) are to be enabled, the CRT calls the function `_cplb_init`, passing the value of `___cplb_ctrl` as a parameter.

The declaration and initialization of the global variable `___cplb_ctrl` is included in the generated CRT if memory protection or caching has been selected through the **Project Wizard**. The default library definition of the variable is used if they are not overridden by a declaration in user code. Refer to "Caching and Memory Protection" on page 1-373.

Lower Processor Priority

The CRT lowers the process priority to the lowest supervisor mode level (`IVG15`). It first raises `IVG15` as an event, but this event cannot be serviced while the processor remains at the higher priority level of `Reset`.

The CRT sets `RETI` to be the `still_interrupt_in_ipend` label, at which there is an `RTI` instruction, and is the next instruction to be executed. This results in all bits representing interrupts of a higher priority than `IVG15` being cleared. In normal circumstances, this would include only the reset interrupt, but occasionally this may not be the case (for example, if the program is restarted during an ISR).

The pending `IVG15` interrupt is now allowed to proceed, and the handler set up earlier in the CRT (at the label `supervisor_mode`) is executed. Thus, execution flows from the "return" from `Reset` level to the `supervisor_mode` label, while changing processor mode from the highest supervisor level to the lowest supervisor level.

> ⓘ If other events are enabled (memory system exceptions or other events installed via your own version of the default handlers stub), they could be taken between the return from `Reset` and entering `IVG15`. Therefore, the remaining parts of the CRT may not execute when event handlers are triggered.

The CRT's first action after entering `IVG15` is to re-enable the interrupt system so that other higher-priority interrupts can be processed.

Mark Registers

The `UNASSIGNED_FILL` value is written into `R2-R7` registers and `P0-P5` registers if the **Project Wizard** option "**Initialize data registers to a known**

value" is selected (or if the UNASSIGNED_FILL macro is defined when rebuilding the default CRT source).

Terminate Stack Frame Chain

Each stack frame is pointed to by the frame pointer and contains the previous values of the frame pointer and RETS. The CRT pushes two instances of UNASSIGNED_VAL onto the stack, indicating that there are no further active frames. The C++ exception support library uses these markers to determine whether it has walked back through all active functions without finding one with a catch for the thrown exception.

Again, the CRT allocates twelve bytes for outgoing parameters of functions that will be called from the CRT.

Profiler Initialization

If profiling is selected (via the **Project Wizard** option "**Enable Profiling**"), the CRT initializes the instrumented-code profiling library by calling monstartup. This routine zeroes all counters and ensures that no profiling frames are active. The instrumented-code profiling library uses stdio routines to write the accumulated profile data to stdout or to a file.

Instrumented-code profiling is specified with the -p, -p1, and -p2 compiler switches. (See "-p[1|2]" on page 1-65.) These are added to the compilation options if necessary by the **Project Wizard**. If any of the object files were compiled to include this profiling, the prelinker detects this and sets link-time macros, which selects a profiler-enabled pre-built CRT object (if the **Project Wizard** is not in use).

C++ Constructor Invocation

The ___ctorloop function runs all of the global-scope constructors, and is always called from the **Project Wizard**-generated CRT and from the C++ enabled pre-built CRTs (which the .ldf file selects if a C++ compiled object has been detected).

For more information, see "Constructors and Destructors of Global Class Instances" on page 1-419.

Multi-Threaded Applications

The CRT can be built to work in a multi-threaded environment. The `_ADI_THREADS` macro guards the code suitable for multi-threaded applications.

Argument Parsing

The `__getargv` function is called to parse any provided arguments (normally an empty list) into the `__Argv` global array. This function returns the number of arguments found which, along with `__Argv`, form the `argc` and `argv` parameters for `_main`. Within the default CRT source, if `FIOCRT` is not defined, `argc` is set to zero and `argv` is set to an empty list, statically defined within the CRT.

Calling _main and _exit

The `_main` function is called, using the `argc` and `argv` just defined. Embedded programs are not expected to return from `_main`, but many legacy and non-embedded programs do. Therefore, the return value from `_main` is immediately passed to `_exit` to gracefully terminate the application. `_exit` is not expected to return.

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 (on page 1-87) 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.

> 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" on page 1-310 and "Placement Support Keyword (section)" on page 1-192), or by altering the default destination section for generated code (see "#pragma section/#pragma default_section" on page 1-310 and "-section" on page 1-72).

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).

Using Memory Sections

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 *VisualDSP++ 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` option (on page 1-53). 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" on page 1-435 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 C run-time library includes four functions:

```
malloc()    calloc()    realloc()    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.

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`.

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 at link-time 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()`. (See "Defining Heaps at Link-Time".) Be sure to pass the correct identifier to each function.

Defining Heaps at Link-Time

Link-time heaps are defined in the `heaptab.s` file in the library, and their start address, length, and `userid` are held in three 32-bit words. The heaps are in a table called "`_heap_table`". This table must contain the default heap (`userid 0`) first and must be terminated by an entry that has a base address of zero.

The addresses placed into this table can be literal addresses, or they can be symbols that are resolved by the linker. The default heap uses symbols generated by the linker through the `.ldf` file.

The `_heap_table` table can live in constant memory. It is used to initialize the run-time heap structure, `___heaps`, when the first request to a heap is made. When allocating from any heap, the library initializes `___heaps` using the data in `_heap_table`, and sets `___nheaps` to be the number of available heaps.

Because the allocation routines use heap indices instead of heap user IDs, a heap installed in this fashion must have its `userid` mapped into an index before it can be used explicitly:

```
int _heap_lookup(int userid);    // returns index
```

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)

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 platform.

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.

The lengths of heaps should be multiples of powers of two for most efficient space usage. The heap allocator works in block sizes such as 256, 512, or 1024 bytes.

For C++ compliance, calls to `malloc` and `calloc` with a size of 0 will allocate a block of size 1.

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.

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,

```
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)
```

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` was 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 the 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 index to the `new` operator. There is no need to pass the heap index to the `delete` operator as the information is not required when the memory is released.

The routines are used as in the example below.

```
#include <heapnew>

char *alloc_string(int size, int heapidx)
{
     char *retVal = new(heapidx) 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 reinitialize 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.

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 for every function call (especially for library calls) and for any possible interrupt.

The dedicated registers are `SP`, `FP`, and `L0-L3`.

- `SP` and `FP` are the stack pointer and the frame pointer registers, respectively. The compiler requires that both point to valid 4-byte aligned addresses within the stack section.
- 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.

Call-Preserved Registers

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. The registers are:

`P3-P5`

`R4-R7`

Scratch Registers

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.

Table 1-42 lists the scratch registers, supplying notes when appropriate.

Table 1-42. Scratch Registers

Scratch Register	Notes
`P0`	Used as the aggregate return pointer
`P1-P2`	
`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`	

Table 1-42. Scratch Registers (Cont'd)

Scratch Register	Notes
LT0-LT1	
ASTAT	Including CC
A0-A1	
I0-I3	
B0-B3	
M0-M3	

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 reached, 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.

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, but must be saved and restored. The stack registers include SP (stack pointer) and FP (frame pointer).

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.

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.

Figure 1-3 shows an example section of a run-time stack.

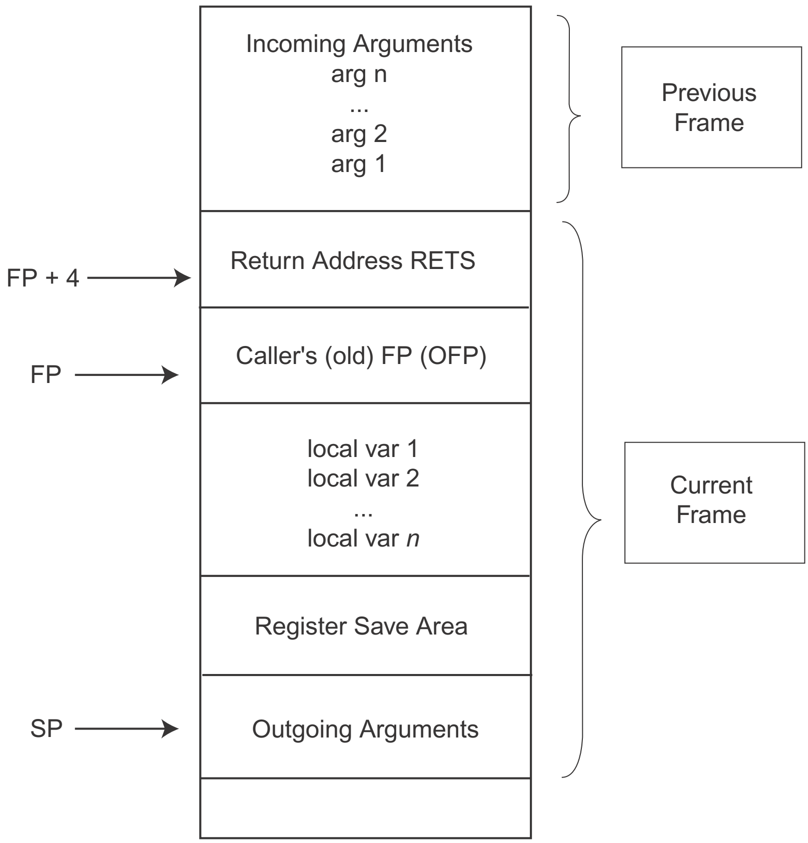


Figure 1-3. Example Run-Time Stack

In Figure 1-3, the currently executing routine, `Current()`, was 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()`.

Stack usage for passing any or all of a function's arguments depends upon the number and types of parameters to the function.

As you write assembly routines, note that operations to restore stack and frame pointers are the responsibility of the called function.

To enter and perform a function, follow this sequence of steps:

- **Linking Stack Frames** – The return address and the caller's `FP` are saved on the stack, and `FP` is set pointing to the beginning of the new (callee) stack frame. `SP` is decremented to allocate space for local variables and compiler temporaries.
- **Register Saving** – Any registers that the function needs to preserve are saved on the stack frame, and `SP` is set pointing to the top of the stack frame.

At the end of the function, these steps must be performed:

- **Restore Registers** – Any registers that had been preserved are restored from the stack frame, and `SP` is set pointing to the top of the stack frame.
- **Unlinking Stack Frame** – The frame pointer is restored from the stack frame to the caller's `FP`, `RETS` is restored from the stack frame to the return address, and `SP` is set pointing to the top of the caller's stack frame.

A typical function prologue would be:

```
LINK    16;
[--SP]=(R7:4);
SP += -16;
[FP+8]=R0; [FP+12]=R1; [FP+16]=R2;
```

where:

> `LINK   16;`
> is a special linkage instruction that saves the return address and the frame pointer, and updates the stack pointer to allocate 16 bytes for local variables.

`[--SP]=(R7:4);`
allocates space on the stack and saves the registers in the save area.

`SP += -16;`
allocates space on the stack for outgoing arguments. Always allocate at least 12 bytes on the stack for outgoing arguments, even if the function being called requires less than this.

`[FP+8]=R0; [FP+12]=R1; [FP+16]=R2;`
saves the argument registers in the argument area.

A matching function epilogue would be:

```
SP += 16;
P0=[FP+4];
(R7:4)=[SP++];
UNLINK;
JUMP (P0);
```

where:

`SP += 16;`
reclaims the space on the stack that was used for outgoing arguments.

`P0=[FP+4]`
loads the return address into register `P0`.

`(R7:4)=[SP++];`
restores the registers from the save area and reclaims the area.

`UNLINK;`
is a special instruction that restores the frame pointer and stack pointer.

`JUMP (P0);`
returns to the caller.

"Transferring Function Arguments and Return Value" on page 1-439 provides additional details on function call requirements.

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. This section describes:

- "Passing Arguments" on page 1-439
- "Passing a C++ Class Instance" on page 1-441
- "Return Values" on page 1-441

Passing Arguments

The details of argument passing are most easily understood in terms of a conceptual argument list. This is a list of words on the stack. Double arguments are placed starting on the next available word in the list, as are structures. Each argument appears in the argument list exactly as it would in storage, and each separate argument begins on a word boundary.

The actual argument list is like the conceptual argument list except that the contents of the first three words are placed in registers `R0`, `R1`, and `R2`. Normally, this means that the first three arguments (if they are integers or pointers) are passed in registers `R0` to `R2` with any additional arguments being passed on the stack.

If any argument is greater than one word, it occupies multiple registers. The caller is responsible for extending any `char` or `short` arguments to 32-bit values.

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.

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` will pass 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

If a function returns a short or a char, the callee is responsible for sign- or zero-extending the return value into a 32-bit register. So, for example, a function that returns a signed short must sign-extend that short into `R0`.

Similarly, a function that returns an unsigned char must zero-extend that unsigned char into `R0`.

- For functions returning aggregate values occupying fewer than or equal to 32 bits, the result is returned in `R0`.
- For aggregate values occupying greater than 32 bits, and fewer than or equal to 64 bits, the result is returned in register pair `R0`, `R1`.
- For functions returning aggregate values occupying more than 64 bits, the caller allocates the return value object on the stack and the address of this object is passed to the callee as a hidden argument in register `P0`.

Table 1-43 provides examples of passed parameters.

Table 1-43. 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`

Table 1-43. Examples of Parameter Passing (Cont'd)

Function Prototype	Parameters Passed as	Return Location

`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]`	

`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)`

Using 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 Table 1-44 and are shown in Figure 1-4 on page 1-449 and Figure 1-5 on page 1-450.

Table 1-44. 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
`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
`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

Table 1-44. Data Storage Formats and Data Type Sizes (Cont'd)

Type	Bit Size	Number Representation	`sizeof` returns
`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

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`.

The native fixed-point types `fract` and `accum` are not available in C++. In C, they are available only when the `stdfix.h` header file is included.

The `fract16` and `fract32` are not actually intrinsic data types—they are `typedefs` to `short` and `long`, respectively. In C, you need to use built-in functions to do basic arithmetic. (See "Fractional Value Built-In Functions in C++" on page 1-232.) You cannot do `fract16*fract16` and get the right result. In C++, for `fract` data, the classes "`fract`" and "`shortfract`" define the basic arithmetic operators, while in C, 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 C and C++ standards which require that the `double` type supports at least 10 digits of precision.

The `-double-size-64` switch (on page 1-34) 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 (on page 1-34), 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. Table 1-45 shows the results for the various permutations:

Table 1-45. Use of the -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 (on page 1-34), 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 (Figure 1-4 on page 1-449) 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.

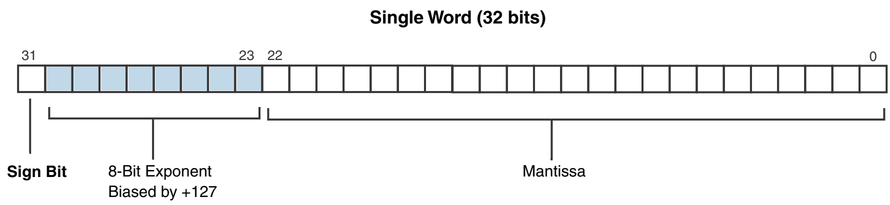


Figure 1-4. Data Storage Format for Float and Double Types

In Figure 1-4, the single word (32-bit) data storage format equates to:

$$-1^{Sign} \times 1.Mantissa \times 2^{(Exponent - 127)}$$

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 (Figure 1-5 on page 1-450) 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.

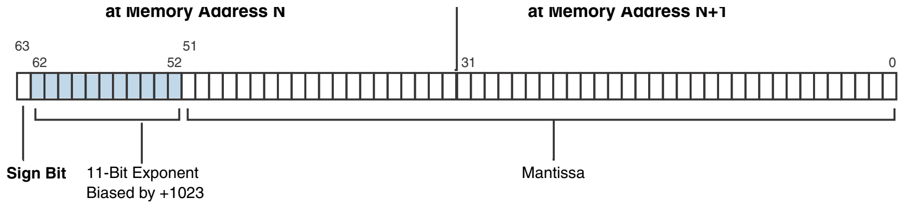


Figure 1-5. Double-Precision IEEE Format

In Figure 1-5, the two-word (64-bit) data storage format equates to:

$$-1^{Sign} \times 1.Mantissa \times 2^{(Exponent - 1023)}$$

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.

Variants of IEEE Floating-Point Support

The Blackfin compiler supports two variants of IEEE floating-point support. These variants are implemented in terms of two alternative emulation libraries, selected at link-time.

The two alternative emulation libraries are:

- The default IEEE floating-point library
  It is a high-performance variant, which relaxes some of the IEEE rules in the interest of performance. This library assumes that its

inputs will be value numbers, rather than `Not-a-Number` values. This library can also be selected explicitly via the `-fast-fp` switch (on page 1-38).

- An alternative IEEE floating-point library
  It is a strictly-conforming variant, which offers less performance, but includes all the input-checking that has been relaxed in the default library. The strictly-conforming library can be selected via the `-ieee-fp` switch (on page 1-45).

The default `.ldf` file links in the appropriate archive(s), depending on the setting of the link-time macro `IEEEFP`. If the `-ieee-fp` switch has been specified, the compiler defines the macro and the `.ldf` file links the application against the non-default, IEEE-conforming library. Conversely, if the link-time macro `IEEEFP` is not defined, then the default `.ldf` file arranges for the application to be linked against the default, high-performance, floating-point archives.

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" on page 1-104.

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 Figure 1-6 on page 1-452.

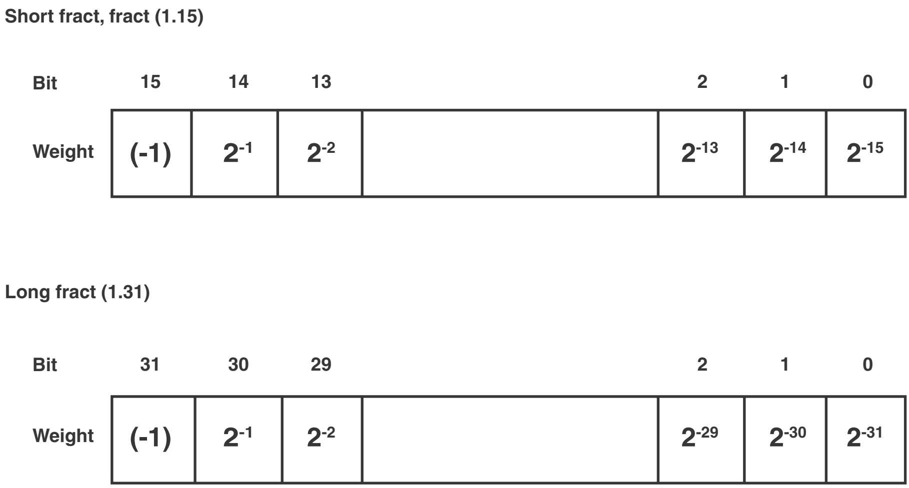


Figure 1-6. Data Storage Format for `short fract`, `fract`, and `long fract`

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 Figure 1-7 on page 1-453.

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`

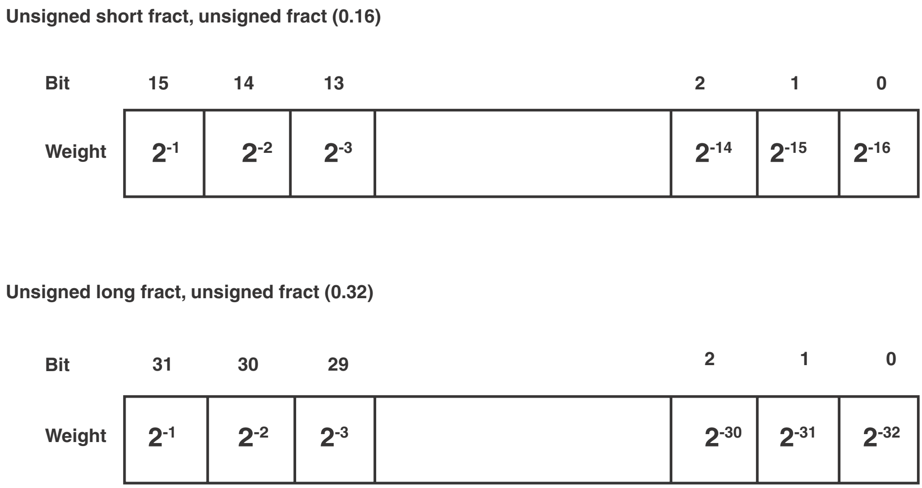


Figure 1-7. Data Storage Format for `unsigned short fract`, `unsigned fract`, and `unsigned long fract`

`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 Figure 1-8 on page 1-454.

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`.

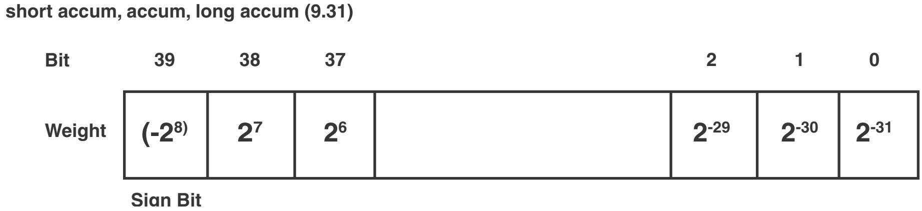


Figure 1-8. Data Storage Format for `short accum`, `accum`, and `long accum`

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 Figure 1-9.

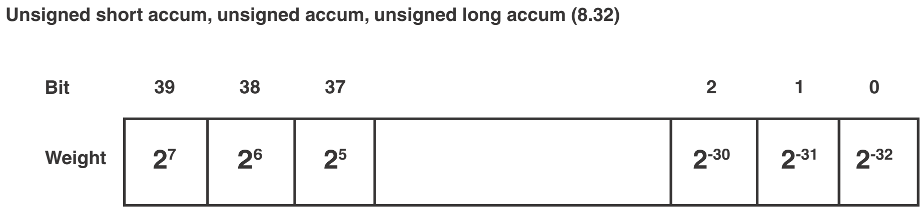


Figure 1-9. Data Storage Format for `unsigned short accum`, `unsigned accum`, and `unsigned long accum`

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`.

```
typedef short fract16;

typedef long  fract32;
```

The `fract` data representation is shown in Figure 1-10

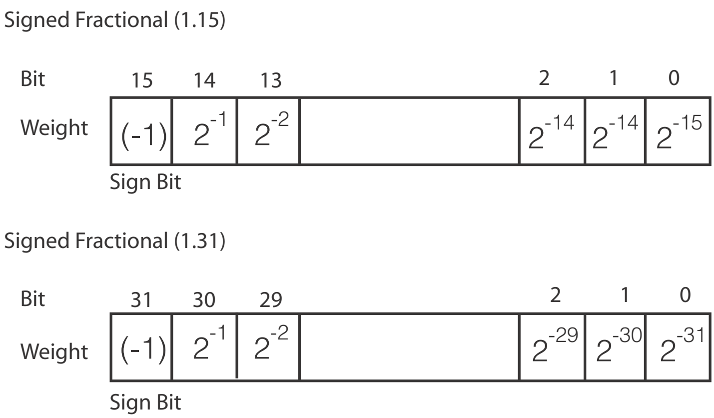


Figure 1-10. Data Storage Format for fract16 and fract32

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.

C/C++ and Assembly Interface

This section describes how to call assembly language subroutines from within C/C++ programs, and how to call C/C++ functions from within assembly language programs. Before attempting to perform either of these operations, familiarize yourself with the information about the C/C++ run-time model (including details about the stack, data types, and how arguments are handled) in "C/C++ Run-Time Model and Environment" on page 1-408. At the end of this reference, a series of examples demonstrate how to mix C/C++ and assembly code.

This section describes:

- "Calling Assembly Subroutines From C/C++ Programs" on page 1-456
- "Calling C/C++ Functions From Assembly Programs" on page 1-459
- "Exceptions Tables in Assembly Routines" on page 1-462

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.

The compiler prefaces the name of any external entry point with an underscore. Therefore, declare your assembly language subroutine's name with a leading underscore.

The run-time model defines some registers as *scratch* registers and others as *preserved* or *dedicated* registers. Scratch registers can be used within the assembly language program without worrying about their previous contents. If more room is needed (or an existing code is used) and you wish to use the preserved registers, you *must save* their contents and then *restore* those contents before returning.

Use the dedicated or stack registers for their intended purpose only; the compiler, libraries, debugger, and interrupt routines depend on having a stack available as defined by those registers.

The compiler also assumes the machine state does not change during execution of the assembly language subroutine.

Do not change any machine modes (for example, certain registers may be used to indicate circular buffering when those register values are nonzero).

The compiler will always align arrays on a 32-bit word boundary, and the compiler will normally use this knowledge when optimizing accesses. It is therefore necessary to ensure that arrays that are defined in assembly code that are accessed in C/C++ code are similarly aligned. This is normally achieved by preceding array definitions in assembly with the `.align 4` assembly directive.

If arguments are on the stack, they are addressed via an offset from the stack pointer or frame pointer. A good way to explore how arguments are passed between a C/C++ program and an assembly language subroutine is to write a dummy function in C/C++ and compile it using the IDDE's **Save temporary files** option (or the `-save-temps` command-line switch).

The following example includes the global volatile variable assignments to indicate where the arguments can be found upon entry to `asmfunc`.

```
// Sample file for exploring compiler interface …
// global variables … assign arguments there just so
// we can track which registers were used
// (type of each variable corresponds to one of arguments):

int global_a;
float global_b;
int * global_p;

// the function itself:

int asmfunc(int a, float b, int * p)
{
   // do some assignments so assembly file will show
       where args are:
   global_a = a;
   global_b = b;
   global_p = p;

   // value gets loaded into the return register:
   return 12345;
}
```

When compiled with the `-save-temps` and `-no-annotate -O` switches, the following code is produced.

```
   .section program;
   .align 2;
_asmfunc:
       P0.L = .epcbss;
       P0.H = .epcbss;
       [P0+ 0] = R0;
```

```
        R0 = 0x1234 (X);
        [P0+ 4] = R1;
        [P0+ 8] = R2;
        RTS;

._asmfunc.end:
    .global _asmfunc;
    .type _asmfunc,STT_FUNC;

    .section data1;

    .align 4;
.epcbss:
        .byte _global_a[4];
        .global _global_a;
        .type _global_a,STT_OBJECT;
        .byte _global_b[4];
        .global _global_b;
        .type _global_b,STT_OBJECT;
        .byte _global_p[4];
        .global _global_p;
        .type _global_p,STT_OBJECT;
.epcbss.end:
```

Calling C/C++ Functions From Assembly Programs

You may want to call C/C++ callable library and other functions from within an assembly language program. As discussed in "Calling Assembly Subroutines From C/C++ Programs" on page 1-456, you may want to create a test function to do this in C/C++, and then use the code generated by the compiler as a reference when creating your assembly language program and the argument setup. Using volatile global variables may help clarify the essential code in your test function.

The run-time model defines some registers as *scratch* registers and others as *preserved* or *dedicated*. The contents of the scratch registers may be changed without warning by the called C/C++ function. If the assembly language program needs the contents of any of those registers, you *must save* their contents before the call to the C/C++ function and then *restore* those contents after returning from the call.

Use the dedicated registers for their intended purpose only; the compiler, libraries, debugger, and interrupt routines depend on having a stack available as defined by those registers.

Preserved registers can be used; their contents are not changed by calling a C/C++ function. The function always saves and restores the contents of preserved registers if they are going to change.

If arguments are on the stack, they are addressed via an offset from the stack pointer or frame pointer. Explore how arguments are passed between an assembly language program and a function by writing a dummy function in C/C++ and compiling it with the `save temporary files` option. (See the `-save-temps` switch on page 1-72.) By examining the contents of volatile global variables in a `*.s` file, you can determine how the C/C++ function passes arguments, and then duplicate that argument setup process in the assembly language program.

The stack must be set up correctly before calling a C/C++ callable function. If you call other functions, maintaining the basic stack model also facilitates the use of the debugger. The easiest way to do this is to define a C/C++ main program to initialize the run-time system; maintain the stack until it is needed by the C/C++ function being called from the assembly language program; and then continue to maintain that stack until it is needed to call back into C/C++. However, ensure that the dedicated registers are correct. You do not need to set the `FP` prior to the call; the caller's `FP` is never used by the recipient.

The assembly interface requires all calling functions to reserve stack space for the first twelve bytes (`R0`-`R2`) of parameter space for a callee, even when the callee does not require that much space. In VisualDSP++ 5.0, the compiler makes increased use of this stack space to store temporary values, if it does not find that the space is needed for other purposes (such as storing the register-based parameter itself). Therefore, all assembly functions that call compiled functions must follow the correct procedure; with VisualDSP++ 5.0, the compiler makes more efficient use of stack space, but there is a corresponding risk that functions that violate the ABI may find that live values are corrupted in the process.

If you call other functions, maintaining the basic stack model also facilitates the use of the debugger. The easiest way to do this is by defining a C/C++ main program to initialize the run-time system, maintaining the stack until it is needed by the C/C++ function being called from the assembly language program, and then continuing to maintain that stack until it is needed to call back into C/C++. However, ensure that the dedicated registers are correct. You do not need to set the `FP` prior to the call; the caller's `FP` is never used by the recipient.

Using Mixed C/C++ and Assembly Naming Conventions

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.

To name an assembly symbol that corresponds to a C symbol, add an underscore prefix to the C symbol name when declaring the symbol in assembly. For example, the C symbol `main` becomes the assembly symbol `_main`. C++ global symbols are usually "*mangled*" to encode the additional type information. Declare C++ global symbols using `extern "C"` to disable the mangling.

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.

Table 1-46 shows several examples of the C/C++ and assembly interface naming conventions.

Table 1-46. C/C++ Naming Conventions for Symbols

In the C/C++ Program	In the 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:`

Exceptions Tables in Assembly Routines

Assembly routines that both call C++ functions and are called by C++ functions and require exceptions thrown by callees to be caught by callers must be provided with a "function exceptions table" to enable the run-time library to restore registers to the values they held on entry to the routine.

The assembly routine must allocate a stack frame using `FP` and `SP` as described in "Managing the Stack" on page 1-435. On entry to the assembly routine, call-preserved registers (on page 1-433) 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 Table 1-48 on page 1-464.

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 Table 1-47.

Table 1-47. Function Exceptions Table

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 Table 1-48, 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 = 1u << (r%32);
if (bitset[wrd] & bit)
    /* register r was saved */
```

Table 1-48. 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
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

Table 1-48. Function Exception Table Register Numbers (Cont'd)

Register	Bit Number	Bytes taken in save area if saved
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;
```

Compiler C++ Template Support

The compiler provides template support C++ templates as defined in the ISO/IEC 14882:2003 C++ standard.

Template Instantiation

Templates are instantiated automatically during compilation using a linker feedback mechanism. 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.

Implicit Instantiation

The compiler uses a method called *implicit instantiation*, which is common practice. It results in having both the specification and definition available at the point of instantiation.

> Implicit instantiation does not conform to the ISO/IEC 14882:2003 C++ standard, and does not work with exported templates. Implicit instantiation is enabled by default. It can be disabled via the `-no-implicit-inclusion` switch on page 1-89.

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 will instruct the compiler to implicitly include the corresponding "`.cpp`" file containing the definitions of the template.

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`" will 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 (on page 1-88) can be used. (For example, the file may contain macro guarded sections of code.) This may result in more time required to generate dependencies.

Exported Templates

The compiler supports the `export` keyword, which provides an alternative implementation for templates. 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 two files are separate translation units; one is not included in the other. This allows the two functions `print()` to coexist (with external linkage).

The automatic instantiation of exported templates is similar to that of regular (included) templates. An instantiation of an exported template involves at least two translation units: one that requires the instantiation, and one that contains the template definition.

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.

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 will occur. 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]' refer-
enced 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 will 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. Explicitly compiling the `.cpp` files that would normally have been implicitly included and adding them to the final link is normally all that is needed to fix this.

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.

This section describes:

- "Automatically-Applied Attributes" on page 1-472
- "Default LDF Placement" on page 1-474
- "Sections Versus Attributes" on page 1-475
- "Using Attributes" on page 1-476

For more information, see "Library Attributes" in Chapter 3, C/C++ Run-Time Library.

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 (on page 1-52).

Figure 1-11 shows a content attribute tree.

The `Content` attribute can be used to map binary objects according to their kind of content, as show in Table 1-49.

Table 1-49. 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.

Table 1-49. Interpreting Values of the `Content` Attribute (Cont'd)

`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 (on page 1-31) and its effects).
`Empty`	The binary object contains neither functions nor global data.

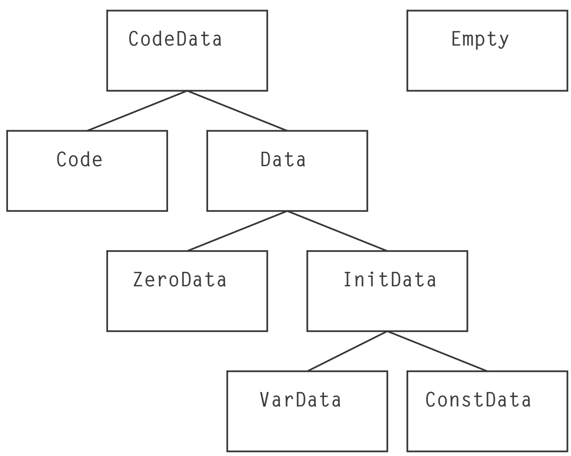


Figure 1-11. Content Attributes

Default LDF Placement

The default `.ldf` file is written in such manner that the order of preference for putting an object in section `data` 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 will be 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. Table 1-50 shows the numerical values used by the run-time library for each corresponding `prefersMem` attribute value.

Table 1-50. Values for prefersMemNum Attribute

prefersMem Attribute Value	prefersMemNum Attribute Value
internal	30
any	50
external	70

Sections Versus Attributes

File attributes and section qualifiers (on page 1-192) 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` (on page 1-314)
- Use the `-file-attr` switch (on page 1-38)

The run-time libraries have attributes associated with the objects in them. For more information, see "Library Attributes" in Chapter 3, C/C++ Run-Time Library.

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")},
      $OBJECTS{ 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 Chapter 3, C/C++ Run-Time Library.

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 switches test.c
```

Both methods will 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.

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.

2 ACHIEVING 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" on page 2-3
  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" on page 2-33
  describes the `expected_true` and `expected_false` built-in functions, which control the compiler's optimization of conditional branches.

- "Loop Guidelines" on page 2-38
  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" on page 2-49
  discusses ways to manipulate fixed-point and fractional data.

- "Using Built-In Functions in Code Optimization" on page 2-54
  describes how to use built-in functions to efficiently use low-level features of the processor hardware while programming in C.

- "Smaller Applications: Optimizing for Code Size" on page 2-57
  provides tips and techniques about optimizing the application for full performance and for space.
- "Using Pragmas for Optimization" on page 2-60
  describes how to use pragmas to finely tune source code.
- "Useful Optimization Switches" on page 2-70
  lists compiler switches useful during the optimization process.
- "How Loop Optimization Works" on page 2-70
  introduces concepts used in loop optimization.
- "Assembly Optimizer Annotations" on page 2-96
  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" on page 2-135
  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 ends with 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 may be possible to improve. These are commented in the code as "`GOOD`" and "`BAD`", respectively.

General Guidelines

This section contains:

- "How the Compiler Can Help" on page 2-4
- "Data Types" on page 2-15
- "Getting the Most From IPA" on page 2-21
- "Indexed Arrays Versus Pointers" on page 2-27
- "Using Function Inlining" on page 2-28
- "Using Inline asm Statements" on page 2-30
- "Memory Usage" on page 2-31

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

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 VisualDSP++ IDDE) is for non-optimized compilation in order to assist programmers in diagnosing problems with their initial coding. The optimizer is enabled in VisualDSP++ by selecting the **Enable optimization** check box on the **Project Options : Compile** page or by using the `-O` switch (on page 1-60). A "release" build from within VisualDSP++ 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 the following diagnostics, which may save time and effort in characterizing source-related problems:

- Warnings and remarks (on page 2-6)
- Assembly annotations (on page 2-7)

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 `volatile` qualifier (on page 2-14) 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.

Warnings 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 (on page 1-79) or as a class, with the `-w` switch (on page 1-80), disabling all warnings and remarks. However, disabling warnings is inadvisable until each instance has been investigated for problems.

A typical warning involves 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 (on page 1-80) or the **Enable remarks** option (**Project : Compile : Warning** page of **Project Options** dialog box).

A typical remark involves a variable being declared, but never used.

A remark may be promoted to a warning through the `-Wwarn` switch (on page 1-79). Remarks and warnings may be promoted to an error through the `-Werror` switch (on page 1-79).

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.

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` (on page 1-338).

Assembly Annotations

By default, the compiler emits annotations that are embedded in the generated assembly code. Annotations can be used to find out why the compiler has generated code in a particular manner.

For more information, see "Assembly Optimizer Annotations" on page 2-96.

Using the Statistical Profiler

Tuning an application begins with identifying areas of the application that are most frequently executed, where improvements would provide the largest gains. The VisualDSP++ statistical profiler provides an easy way to find these areas. VisualDSP++ Help explains how to use the profiler in detail.

The advantage of statistical profiling is that it is completely unobtrusive. Other forms of profiling insert instrumentation into the code, disturbing the original optimization, code size, and register allocation.

The best methodology is usually to compile with both optimization and debug information generation enabled. You can then obtain a profile of the optimized code while retaining function names and line number information. This gives you accurate results that correspond directly to the C/C++ source. Note that the compiler optimizer may have moved code between lines.

If you build your application optimized but without debug information generation, the profile will obtain statistics that relate directly to the assembly code. This kind of profile provides the most precise view of your application but not usually the easiest to use because you must relate assembly lines to the original source. Do not strip out function names when linking, since keeping function names means you can scroll through the assembly window to instructions of interest.

In complex code, you can locate the exact source lines by counting the loops, unless they are unrolled. Looking at the line numbers in the assembly file may also help. (Use the `-save-temps` switch to retain compiler generated assembly files, which have the `.s` filename extension.) The compiler optimizer may have moved code around, so that it does not appear in the same order as in your original source.

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 that PGO is supported in the simulator only.

An example application that demonstrates how to use PGO is in "Example of Profile-Guided Optimization" on page 2-37.

Using Profile-Guided Optimization With a Simulator

The PGO process is illustrated in Figure 2-1.

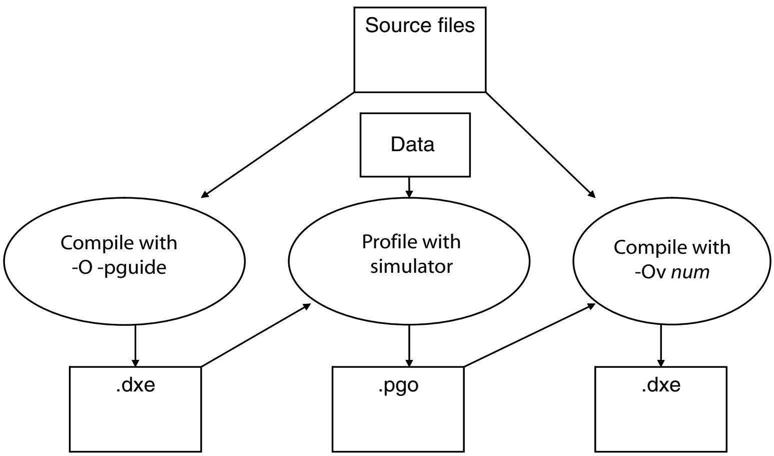


Figure 2-1. PGO Process

1. Compile the application with the `-pguide` switch (on page 1-67) or **Prepare application to create new profile** option. This creates an executable file containing the necessary instrumentation for gathering profile data. For best results, use the **Enable optimization** option/`-O` switch (on page 1-60) or **Interprocedural analysis** option/`-ipa` (on page 1-47) switch.

2. Gather the profile. Presently, this may only be done using a simulator. Run the executable 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 a file with the extension `.pgo`.

3. Recompile the application using this gathered profile data. Place the `.pgo` file on the command line. Optimization should also be enabled at this stage.

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:

```
ccblkfn -O f2.c -o f2.doj prof1.pgo

ccblkfn -o prog.dxe f1.asm f2.doj prof2.pgo
```

See also "Using PGO in Function Profiling" on page 2-37.

Using Profile-Guided Optimization With Non-Simulatable Applications

It may not be possible to run a complex application in its entirety in a simulation session (for example, if peripherals not modeled by the simulator are used). It may, however, still be possible to use PGO as follows.

1. If the application is structured in a modular fashion, it will be possible to extract the core performance-critical algorithm from the application.
2. Create a "wrapper" project, which can be run under simulation that drives input values into the core algorithm, replacing the portions of the application that can not be run under simulation. This project can be used to generate PGO information, which can subsequently be used to optimize the full application. As described earlier, it is essential that the input values are representative of real data to achieve best performance.
3. Leave as much of the core algorithm unmodified as possible, keeping file and function names the same. The `.pgo` files generated from execution of the wrapper project can then be used to optimize the same functions in the full application by including the `.pgo` files in the full application build.

When compiling with a `.pgo` file, the compiler emits a warning and ignores the data for a function if it detects the function has changed from when the PGO data was generated. Therefore, any functions that you do modify to get the algorithm to work properly outside the application will not benefit from the profile information.

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 multiprocessor environment. In such circumstances, the typical behaviors of each instance in the application might differ. Identify the separate instances so that they can be profiled separately and optimized accordingly.

The `-pgo-session` switch (on page 1-67) (or **PGO session name** option) 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 100` switch. The `-Ov n` switch (on page 1-61) is discussed further along with optimization for space in "Smaller Applications: Optimizing for Code Size" on page 2-57.

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). The programmer should ensure that the profile data used for optimization remains accurate.

For more details on PGO, refer to "Optimization Control" on page 1-95.

An example application demonstrates how to use PGO in "Example of Profile-Guided Optimization" on page 2-37.

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 (on page 1-47) 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 (on page 1-71). To see the assembly file when IPA is enabled, use the `-save-temps` switch (on page 1-72) 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 `__builtin_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 __builtin_aligned" on page 2-24 and "Using Pragmas for Optimization" on page 2-60.

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 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

Table 2-1 shows compiler-supported scalar data types.

Table 2-1. Scalar Data Types

Data Type	Description
Single-Word Fixed-Point 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

Table 2-1. Scalar Data Types (Cont'd)

Data Type	Description
`unsigned long`	32-bit unsigned integer
Fixed-Point Data Types: Native and Emulated Arithmetic	
`short fract` (C only)	16-bit signed fractional
`fract` (C only)	16-bit signed fractional
`long fract` (C only)	32-bit signed fractional
`unsigned short fract` (C only)	16-bit unsigned fractional
`unsigned fract` (C only)	16-bit unsigned fractional
`unsigned long fract` (C only)	32-bit unsigned fractional
`short accum` (C only)	40-bit signed fixed-point
`accum` (C only)	40-bit signed fixed-point
`long accum` (C only)	40-bit signed fixed-point
`short unsigned accum` (C only)	40-bit unsigned fixed-point
`unsigned accum` (C only)	40-bit unsigned fixed-point
`long unsigned accum` (C only)	40-bit unsigned fixed-point
Double-Word Fixed-Point Data Types: Emulated Arithmetic	
`long long`	64-bit signed integer
`unsigned long long`	64-bit unsigned integer

Table 2-1. Scalar Data Types (Cont'd)

Data Type	Description
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 in C mode 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 these data types is best done by using the built-in functions, described in "Using System Support Built-In Functions" on page 2-54.

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)" on page 1-286 for more 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” on page 2-15.

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.

For example, such operations in a loop can prevent the compiler from using efficient zero-overhead hardware loop instructions. Also, calling the library to perform the required operation can change values held in scratch registers before the call, so the compiler has to generate more stores and loads from the data stack to keep values required after the call returns. Avoid emulated arithmetic operators where possible, especially in loops.

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” on page 2-21
- “Word-Aligning Your Data” on page 2-23
- “Using __builtin_aligned” on page 2-24
- “Avoiding Aliases” on page 2-25

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;        // reassigned
}

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 Your 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 must 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. It may be worth 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 __builtin_aligned

To avoid the need to use IPA to propagate alignment, and for situations when IPA cannot guarantee the alignment (but you can), use the `__builtin_aligned` function to assert the alignment of important pointers, meaning that the pointer points to data that is aligned.

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.
void copy(char *a, char *b) {
   int i;
   __builtin_aligned(a, 4);
   __builtin_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.
void copy(char *a, char *b) {
int i;
__builtin_aligned(a+1, 4);
__builtin_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 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 might hope that the chosen style would not matter to the generated code, but this 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.

ⓘ Try both pointer and indexed styles.

The pointer style introduces additional variables that compete with the surrounding code for resources during the compiler optimizer's analysis. Array accesses, on the other hand, must be transformed to pointers by the compiler, and sometimes this 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 (on page 1-60) or the **Inlining -> Automatic** option, automatically enabling optimization.

ⓘ 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 (on page 1-61) 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.

Using Inline asm Statements

The compiler allows use of inline `asm` statements to insert small sections of assembly into C code.

> 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" on page 1-195.

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" on page 2-54.

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.

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];
```

First, 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, 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, declare arrays with the `section("section_name")` construct preceding a buffer declaration; in this case,

```
section("bank_a1") short a[100];
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.

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 "Bank Qualifiers" on page 1-191 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 ("Compiler Performance Built-In Functions" on page 1-264) the compiler will make 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" on page 2-34
- "Using PGO in Function Profiling" on page 2-37

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 Hardware Required/
     Compiler Features/Branch Prediction
```

The example project, called `branch_prediction`, 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 Table 2-2. 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 2-2. 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. Create a simulator session for the ADSP-BF533 Blackfin processor.
2. Open the `branch_prediction` project.
3. Build the project, load it into the simulator, and execute it. 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.
4. Open the **Project Options** dialog box, and go to the **Preprocessor** area of the **Compile** page.
5. In the **Defines** field, add `EXPRS=1`. Click **OK**.
6. Rebuild and rerun the application. You will receive the same counts from the application, but the cycle counts will be different.
7. 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" on page 1-264 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" on page 1-95 for more details.

Example of Profile-Guided Optimization

Continuing with the same example (on page 2-35), 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.

To use the example, do the following:

1. Create a simulator session for the ADSP-BF533 Blackfin processor.
2. Open the `branch_prediction` project.
3. Open the **Project Options** dialog box, and display the **Preprocessor** area of the **Compile** page.
4. Make sure that the **Defines** field does not include a definition for the `EXPRS` macro. Click **OK**.
5. Via **Tools**, **PGO**, select **Manage Data Sets**. The **Manage Data Sets** dialog box appears.
6. Click **New**. The **Edit Data Set** dialog box appears.
7. In the **Output filename** (`.pgo`) field, enter the path name where the simulator should create the generated execution profile. This path name must have a `.pgo` extension. Click **OK**.

8. Click **OK** again, to close the **Manage Data Sets** dialog box.
9. Via **Tools**, **PGO**, select **Execute Data Sets**. VisualDSP++ will do the following:
   a. Build the application with the `-pguide` switch, which prepares it to gather a profile.
   b. Run the executable in the simulator, using the data sets provided. The profile will be stored in the `.pgo` file you specified.
   c. Rebuild the application with the gathered profile, which selects the branch prediction according to the most-frequently executed paths of control.
   d. Open a window displaying the difference in performance as a result of the profile-based tuning.

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, and the data would influence the gathered profile. For this example, all the input data is embedded in the application source, so the data set is a null set containing no input files.

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" on page 2-39
- "Avoiding Unrolling Loops" on page 2-39

- "Avoiding Loop-Carried Dependencies" on page 2-40
- "Avoiding Loop Rotation by Hand" on page 2-41
- "Avoiding Complex Array Indexing" on page 2-42
- "Inner Loops Versus Outer Loops" on page 2-43
- "Avoiding Conditional Code in Loops" on page 2-43
- "Avoiding Placing Function Calls in Loops" on page 2-44
- "Avoiding Non-Unit Strides" on page 2-45
- "Using 16-Bit Data Types and Vector Instructions" on page 2-46
- "Loop Control" on page 2-47
- "Using the Restrict Qualifier" on page 2-48
- "Avoiding Long Latencies" on page 2-49

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

ⓘ 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

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

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" on page 2-15.

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" on page 2-59.

Loop Control

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++;
}
```

Avoiding Long Latencies

All pipelined machines introduce stall cycles when you cannot execute the current instruction until a prior instruction has exited the pipeline. For example, the Blackfin processor stalls for three cycles on a table lookup. `a[b[i]]` takes four cycles more than expected.

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" on page 2-50
- "Using the Native Fixed-Point Types fract and accum" on page 2-51

- "Using Built-In Functions to Perform Fixed-Point Arithmetic" on page 2-52
- "Using the shortfract and fract Classes in C++" on page 2-53

Using Integer Arithmetic to Encode Fractional Semantics

One way to manipulate fractional data involves the use of multiply-and-shift 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 undesirable 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 the 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 to provide further manipulations on these types. For more information, see "Using Native Fixed-Point Types" on page 1-104.

There are two important restrictions on using these types. Firstly, they are not available when compiling in C++ mode, so C++ code cannot use the native fixed-point types. Secondly, they are not compliant with MISRA, 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 both C and C++ modes, 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 in C++" on page 1-232 and "Full-Precision Accumulator Built-In Functions" on page 1-247.

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`'d.

Using the shortfract and fract Classes in C++

If compiling in C++ mode, the `shortfract` and `fract` classes can be used. Arithmetic on these types using the usual arithmetic operators will obey fractional semantics. For more information, see "Fractional Value Built-In Functions in C" on page 1-196.

> The native fixed-point type `fract` represents a 16-bit fractional value, while the C++ `fract` class represents a 32-bit fractional value.

Like the native fixed-point types `fract` and `accum` (which cannot be used in C++ mode), this style leads to readable code and makes the fractional semantics clear to the compiler. The following example shows this approach being used to write a dot product on fractional data.

```
// GOOD: uses shortfract and fract classes  to implement frac-
tional multiplication
#include <fract>
#include <shortfract>
fract dot_product(shortfract *a, shortfract *b) {
   int i;
   fract sum=0.0r;
   for (i=0; i<100; i++) {
      /* this line is performance critical */
      sum += (fract)a[i] * (fract)b[i];
   }
   return sum;
}
```

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" on page 1-195.

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" on page 2-49.

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" on page 1-195.

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 (on page 1-39). 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.

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
   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
   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" on page 1-256.

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` (on page 1-61), 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 (on page 1-39). 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 on page 1-61. 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 "Optimization Control" on page 1-95.) Without PGO, the execution counts are estimated, based on the depth of loop nesting.

> 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" on page 2-19 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 (on page 2-43), 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.

Listing 2-1. 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 Listing 2-1 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" on page 1-277 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 section describes "Function Pragmas" while "Loop Optimization Pragmas" are described on page 2-65.

Function Pragmas

Function pragmas include `#pragma alloc`, `#pragma const`, `#pragma pure`, `#pragma result_alignment`, and `#pragma regs_clobbered`. The optimization `#pragma optimize_{off|for_speed|for_space|as_cmd_line}` is also useful to control the optimization strategy used for specific functions in the source file.

#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 the buffer returned by the call to `new_buf` does not modify the input buffer `a`. 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;
}
```

Further details on this pragma are in "#pragma result_alignment (n)" on page 1-330. Another, more laborious, way to achieve the same effect is to use `__builtin_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 might 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" on page 1-322.

#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.

The #pragma optimize_as_cmd_line resets the optimization settings to those specified on the ccblkfn command line when the compiler is invoked. Refer to "General Optimization Pragmas" on page 1-297 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 run-time 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)" on page 1-292.

#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.

#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 instruct the compiler to execute four iterations 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" on page 1-296.

#pragma all_aligned

The `all_aligned` pragma is used as shorthand for multiple `__builtin_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 __builtin_aligned to give alignment of a and b.
__builtin_aligned(a, 4);
__builtin_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 `n`th 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 __builtin_aligned to give alignment of a and b.
__builtin_aligned(a+96, 4);
__builtin_aligned(b+96, 4);
for (i=99; i>=0; i--)
   a[i] = b[i];
```

For more information, refer to "#pragma all_aligned" on page 1-288 and "Using __builtin_aligned" on page 2-24.

#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" on page 1-288 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" on page 1-295 for more information.

Useful Optimization Switches

Table 2-3 lists compiler switches useful during the optimization process.

Table 2-3. C/C++ Compiler Optimization Switches

Switch Name	Description
`-const-read-write` on page 1-31	Specifies that data accessed via a pointer to `const` data may be modified elsewhere
`-flags-link -e` on page 1-39	Specifies linker section elimination
`-force-circbuf` on page 1-39	Treats array references of the form `array[i%n]` as circular buffer operations
`-ipa` on page 1-47	Turns on inter-procedural optimization. Implies use of `-O`. May be used in conjunction with `-Os` or `-Ov`.
`-no-fp-associative` on page 1-55	Does not treat floating-point multiply and addition as an associative
`-O` on page 1-60	Enables code optimizations and optimizes the file for speed
`-Os` on page 1-61	Optimizes the file for size
`-Ov num` on page 1-61	Controls speed vs. size optimizations (sliding scale)
`-save-temps` on page 1-72	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

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 reordering 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 recomputed 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 `k`th 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 on-going 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.

> 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" on page 2-124.

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" on page 2-96 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

Table 2-4 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 Table 2-5.

Table 2-4. Base Schedule

Cycle	Instructions
1	I1
2	I2, I3
3	I4, I5
4	I6
5	I7
6	I8

Table 2-5. 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 Table 2-6 repeats every 2 cycles. This repeating pattern, the kernel, represents the modulo-scheduled loop.

Table 2-6. 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)" on page 2-84
- "Minimum Initiation Interval Due to Recurrences (Rec MII)" on page 2-85
- "Stage Count (SC)" on page 2-85
- "Variable Expansion and MVE Unroll" on page 2-87
- "Trip Count" on page 2-92

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 Table 2-6 on page 2-83 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 Table 2-7, where each stage is viewed as a vector of height II=2 of instruction lists (that represent parts of instruction lines).

Table 2-7. 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 Table 2-8. Table 2-9 shows the base schedule that is an instance of the one in Table 2-4 on page 2-82, and Table 2-10 on page 2-88 shows the corresponding modulo schedule with II=2.

Table 2-8. 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 2-9. Base Schedule from Table 2-4 Applied to Instances in Table 2-8

1	t1=[p1++]
2	t2=[p2++],t3=t1+t5
3	t4=t2+1,t5=t1+t3
4	t6=t4*t5
5	t7=t6*t3
6	[p8++]=t7

Table 2-10. 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 Table 2-10: `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 Table 2-11 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 2-11. 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 Table 2-12). 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 Table 2-12.

Table 2-12. 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

A Working 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 "Optimization Control" on page 1-95.
- With the `-sat-associative` command-line switch (on page 1-71). 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>
long fract sp(fract *a, fract *b) {
   int i;
   accum sum=0.0k;
   __builtin_aligned(a, 4);
   __builtin_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 `__builtin_aligned`, although it could also have been propagated with IPA.

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.

Vectorization is only possible in this example because the `-sat-associative` switch enables reordering 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 could 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 (on page 1-71), the `-save-temps` switch (on page 1-72), or by checking the **Project Options->Compile->General->Save temporary files** option in VisualDSP++ IDDE.

ⓘ For more information about the IDDE, refer to VisualDSP++ online Help.

The assembly code generated by the compiler optimizer is annotated with the following information:

- "Global Information" on page 2-97
- "Procedure Statistics" on page 2-99
- "Instruction Annotations" on page 2-103
- "Loop Identification" on page 2-103
- "Vectorization" on page 2-115
- "Modulo Scheduling Information" on page 2-124
- "Warnings, Failure Messages, and Advice" on page 2-130

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.

The assembly output for the examples in this chapter may differ based on optimization flags and the version of the compiler. As a result, you may not be able to reproduce these results exactly.

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.

For instance, if the file `hello.c` is compiled at 11am, on June 28 using the following command line:

```
ccblkfn -O -S hello.c
```

then the `hello.s` file will show:

```
.file "hello.c";
//  Compilation time: Thu Jun 28 11:00:00 2007
//  Compiler options: -O -S
//  Architecture: ADSP-BF532
//  Silicon revision: 0.3
//  Anomalies summary:
//    Disabled: w05_00_0046,w05_00_0048,w05_00_0054, .....
//    Enabled: w05_00_0189,w05_00_0198,w05_00_0202, .....
//    Always on: w05_00_0074,w05_00_0122
```

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:

1. 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.
2. 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.
3. 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` = name of the function inlined.
`line` = line number of the call to `function_name`, in the source file.
`column` = column number of the call to `function_name`, in the source file.
`file_name` = name of the source file calling `function_name`.

Example A (Procedure Statistics)

Consider the following program:

```
int func1(int*);
int func2(int);

int foo(int in)
{
    int loc1 = 20;
    int loc_arr[20];
    loc1 = func1(loc_arr);
    in += func2(loc_arr[loc1]);
    return loc_arr[in];
}
```

The procedure statistics for `foo` are:

```
_foo:
.LN_foo:
.reference _func1;
.reference _func2;
//-------------------------------------------------------
// Procedure statistics:
// Frame size = 96
// Scratch registers used:{R0.L,R0.H,R1.L,R1.H,
//                          P0-P2,ASTAT}
// Call preserved registers used:{R7.L,R7.H,P5,FP,SP,RETS}
// Registers that could be clobbered by function calls:
//   {R0.L,R0.H,R1.L,R1.H,R2.L,R2.H,R3.L,R3.H,
//  P0-P2,I0-I3,B0-B3,M0-M3,A0.W,A0.X,A1.W,A1.X,
//  ASTAT,CC,AQ,LC0-LC1,LT0-LT1,LB0-LB1,
//  RETS,SEQSTAT,SYSCFG,USP}
//----------------------------------------------------------
// line "moo2.c":13
LINK 80;
```

```
.align 2
[--SP] = (R7:7, P5:5);
SP += -12;
R7 = R0;
  ...
```

Notes:

- The frame size is 96 bytes, indicating how much space is allocated on the stack by the function. The frame size includes:
  - 4 bytes for `RETS`
  - 4 bytes for the frame pointer
  - Space allocated by the compiler, for local variables (80 bytes for `loc_arr[20]`)
  - Space required to save any callee-preserved registers (8 bytes, for `R7` and `P5`)
  - Space required for parameters being passed to functions called by this one (none in this case)
- "Scratch registers used" refers to those registers the compiler does not need to save before modifying. In this case, the registers are `R0`, `R1`, `P0`, `P1`, `P2`, and `ASTAT`. This does not include any registers that are modified only by calls to other functions.
- "Call-preserved registers used" refers to those registers which must be saved before modification, and restored afterwards. In this case, the compiler uses `R7` and `P5`, and the saved value for these registers account for 8 bytes of frame size.
- "Registers that could be clobbered by function calls" refers to the union of all the registers that will be modified by the calls to other functions. In this case, the registers are the default scratch register set, modified by calls to `func1` and `func2`.

Example B (Inlining Summary)

This is an example of inlined function reporting.

```
void f4(int n);
__inline void f3(int n)
{
  f4(n);
}

__inline void f2(int n)
{
  while (n--) {
     f3(n);
     f3(2*n);
  }
}
void f1(volatile unsigned int i)
{
  f2(30);
}
```

`f1` inlines the call of `f2`, which inlines the call of `f3` in two places. The procedure statistics for `f1` reports these inlined calls:

```
_f1:
//-------------------------------------------------------------
//  Procedure statistics
. . . . .
//Inlined in _f1:
//   ExampleB.c:16:7'_f2
//        ExampleB.c:11:11'_f3
//        ExampleB.c:10:11'_f3
//-------------------------------------------------------------
. . . . .
```

f1 reports that f2 was inlined at line 16 (column 7) and, implicitly, f1 also inlined the two calls of f3 inside f2.

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 (on page 1-30) 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. Thus

```
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}
```

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" on page 2-104
- "Resource Definitions" on page 2-106
- "File Position" on page 2-110
- "Infinite Hardware Loop Wrappers" on page 2-112

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.
- 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 we show 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.

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. In either 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.

Example C (Loop Identification)

Consider the following example:

```
int bar(int a[10000])
{
   int i, sum = 0;
   for (i = 0; i < 9999; ++i)
```

```
5           sum += (sum + 1);
6       while (i-- < 9999)   /* this loop doesn't get executed */
7           a[i] = 2*i;
8       return sum;
9 }
```

The two loops are accounted for as follows:

```
_bar:
//-----------------------------------------------------------
............... procedure statistics ............
//-----------------------------------------------------------
// Original Loop at "ExampleC.c" line 6 col 3
// Loop structure removed due to constant propagation.
//-----------------------------------------------------------
// line "ExampleC.c":4
        P1 = 9999;
        .align 2
        R0 = 0;
// line 5
        R1 = 1;
// line 4
        LOOP .P34L2L LC0 = P1;
.P34L2:
//------------------------------------------------------------
//   Loop at "ExampleC.c" line 4 col 3
//------------------------------------------------------------
//   This loop executes 1 iteration of the original loop
//   in estimated 2 cycles.
//------------------------------------------------------------
//   This loop's resource usage is:
//        16-bit Instruction  used   4 out of    4 (100.0%)
//        32-bit Instruction  used   2 out of    2 (100.0%)
//        Group 1             used   2 out of    2 (100.0%)
//-----------------------------------------------------------
        LOOP_BEGIN .P34L2L;
```

```
// line 5
        R2 = R0 + R1;
        R0 = R0 + R2;
// line 4
        LOOP_END .P34L2L;
.P34L13:
//---------------------------------------------------------
//   Part of top level (no loop)
//-----------------------------------------------------------
// line 8
        RTS;
.LN._bar.end:
```

Notes:

- The keywords identifying the two loops are:
  - `for` – Its position is in the file `ExampleC.c`, line 4, column 3.
  - `while` – Its position is in file `ExampleC.c`, line 6, column 3.
- Immediately after the procedure statistics, a message states that the loop at line 6 in the user program was removed. The reason was constant propagation, which in this case realizes that the value of `i` after the first loop is 9999, and that the second loop does not get executed.
- The start of the loop at line 4 is marked in the assembly by the "Loop at ExampleC.c, line 4, column 3" annotation. This annotation follows the loop label `.P34L2`. The loop label `End Loop L2` is used to identify the end of the loop.
- The loop resource information accounts for all instructions and stalls inside the loop. In this particular case, the loop body is executed in two cycles, one instruction for each cycle. Both instructions are single-issue instructions. The compiler has not issued any `NOP` or `MNOP` instructions, so it reports full utilization.

File Position

As seen in Example C, the following file position is given, using the file name, line number, and the column number in that file:

```
"ExampleC.c" line 4 col 6
```

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>.

Example D (Inlining Locations)

Consider the following source code:

```
void f2(int n);
inline void f3(int n)
{
   while(n--)
          f4();
      if (n == 7)
          f2(3*n);
}

inline void f2(int n)
{
   while(n--) {
        f3(n);
        f3(2*n);
   }
}
void f1(volatile unsigned int i)
```

```
22  {
23     f2(30);
24  }
```

The annotations generated for function `f1` is structured as follows:

```
.....
// Inlined in _f1:
//  ExampleD.c:23:5'_f2
//    ExampleD.c:18:7'_f3
//    ExampleD.c:17:7'_f3
//--------------------------------------------------------------
// line "ExampleD.c":22
LINK 0;
.....
.P36L4:
//-------------------------------------------------------------
//    Loop at "ExampleD.c" line 16 col 3 inlined
//    at "ExampleD.c" line 23 col 5
//-------------------------------------------------------------
.....
.P36L7:
//-------------------------------------------------------------
//    Loop at "ExampleD.c" line 8 col 3 inlined at "ExampleD.c"
//    line 17 col 7 inlined at "ExampleD.c" line 23 col 5
//-------------------------------------------------------------
.....
//-------------------------------------------------------------
//    End Loop L7
//-------------------------------------------------------------
.P36L31:
//-------------------------------------------------------------
//    Part of Loop 4, depth 1
//-------------------------------------------------------------
.P36L8:
```

```
// line 10
.....
.P36L15:
//-----------------------------------------------------------------
//    Loop at "ExampleD.c" line 8 col 3 inlined at "ExampleD.c"
//    line 18 col 7 inlined at "ExampleD.c" line 23 col 5
//----------------------------------------------------------------
.....
```

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.

Example E (Hardware Loop Wrappers)

Consider the following example:

```
int pseudo_mod(int l, int r)
{
   while (l > r) {
        l -= r;
   }
```

```
6    return l;
7 }
```

and the code generated for this:

```
CC = R1 < R0;
   if !CC jump .P34L2 ;
   P1 = -1;
.P34L10:
//-------------------------------------------------------------
//   Loop at "ExampleE.c" line 3 col 3
//   (infinite hardware loop wrapper)
//-------------------------------------------------------------
   LOOP .P34L3L LC0 = P1;
.P34L3:
//-------------------------------------------------------------
//   Loop at "ExampleE.c" line 3 col 3
//-------------------------------------------------------------
   LOOP_BEGIN .P34L3L;
// line 4
   R0 = R0 - R1;
// line 3
   CC = R1 < R0;
   if !CC jump .P34L2 ;
.P34L9:
   LOOP_END .P34L3L;
//-------------------------------------------------------------
//   End Loop L3
//-------------------------------------------------------------
.P34L11:
//-------------------------------------------------------------
//   Part of Loop 10, depth 1
//-------------------------------------------------------------
   jump .P34L10;
//-------------------------------------------------------------
```

```
//   End Loop L10
//------------------------------------------------------------
.P34L2:
//------------------------------------------------------------
//   Part of top level (no loop)
//------------------------------------------------------------
// line 6
   RTS;
```

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

Another vectorization-related transformation is unroll and jam. Consider the following function:

```
/* unroll and jam example */
void f_unroll_and_jam(short a[][40], short *restrict c) {
   int i, j;
   __builtin_aligned(a, 4);
   __builtin_aligned(c, 4);
   for (i=0; i<60; i++) {
       short sum=0;
       for (j=0; j<40; j++) {
           sum += a[j][i];
       }
       c[i] = sum;
   }
}
```

The outer loop can be unrolled twice and the result is:

```
void f_unroll_and_jam(short a[][40], short *restrict c) {
   int i, j;
   __builtin_aligned(a, 4);
   __builtin_aligned(c, 4);
   for (i=0; i<60; i+=2) {
       {
         short sum=0;
         for (j=0; j<40; j++) {
             sum += a[j][i];
         }
         c[i] = sum;
      }
      {
         short sum=0;
         for (j=0; j<40; j++) {
             sum += a[j][i+1];
         }
         c[i+1] = sum;
      }
   }
}
```

The two inner loops can be jammed together. We shall assume that we have a `plus_eq2` operation which is a more powerful version of += that can handle two short integers at a time.

The result is:

```
void f_unroll_and_jam(short a[][40], short *restrict c) {
   int i, j;
   __builtin_aligned(a, 4);
   __builtin_aligned(c, 4);
   for (i=0; i<60; i+=2) {
       short sum0=0;
```

```
        short sum1=0;
        for (j=0; j<40; j++) {
            (sum0, sum1) .plus_eq2. (a[j][i], a[j][i+1]);
        }
        (c[i], c[i+1]) = (sum0, sum1);
    }
}
```

Example F (Unroll and Jam)

The assembly-annotated code for the above `f_unroll_and_jam` example is:

```
              M0 =  80 (X):
              LOOP ._P1L2 LC1 = P2;
//  "ExampleF.c" line 8 col 83
              P2 =  39;

.P1L2:
//-----------------------------------------------------------
//  Loop at "ExampleF.c" line 6 col 4
//-----------------------------------------------------------
//  Loop was unrolled for unroll and jam 2 times
//-----------------------------------------------------------
              LOOP_BEGIN ._P1L2;
              I0 = P0 :
              R0 = ROT R1 by 0  || NOP || R2 = [I0++M0];
              LOOP ._P1L4 LC0 = P2;

.P1L4:
//-----------------------------------------------------------
//  Loop at "ExampleF.c" line 8 col 8;
//-----------------------------------------------------------
//  This jammed loop executes 2 iterations of the original loop
//  in 1 cycle.
//  (1 iteration of the inner loop for each of the 2 unrolled
```

```
//  iterations of the outer loop)
//-----------------------------------------------------------
//  This loop's resource usage is:
//    32-bit ALU/MAC used 1 out of 1 (100.0%)
//    Group 1 or 2 used 1 out of 2 ( 50.0%)
//-----------------------------------------------------------
//  Loop was jammed by unroll and jam 2 times
//-----------------------------------------------------------
//  "ExampleF.c" line 9 col 13
              LOOP_BEGIN ._P1L4;
                R0 = R0 +|+ R2 || NOP || R2 = [I0++M0];
//  "ExampleF.c" line 8 col 8
                R0 = R0 + R2;
              LOOP_END ._P1L4;
//-----------------------------------------------------------
//  End Loop L4
//-----------------------------------------------------------

.P1L5:
//-----------------------------------------------------------
//  Part of Loop 2, depth 1
//-----------------------------------------------------------
//  "ExampleF.c" line 9 col 13
                R0 = R0 +|+ R2;
//  "ExampleF.c" line 11 col 8
                [P1++] = R0;
                P0 +=  4;
//  "ExampleF.c" line 6 col 4
              LOOP_END ._P1L2;
//-----------------------------------------------------------
//  End Loop L2
//-----------------------------------------------------------
```

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.

Example G (Loop Flattening):

For instance, the following function

```
void copy_v(int a[][100], int b[][100]) {
   int i,j;
   for (i=0; i< 30; ++i)
      for (j=0; j < 100; ++j)
          a[i][j] = b[i][j];
}
```

is transformed into

```
void copy_v(int a[][100], int b[][100]) {
   int i,j;
   int *p_a = &a[0][0];
   int *p_b = &b[0][0];
   for (i=0; i< 3000; ++i)
       p_a[i] = p_b[i];
}
```

This may further facilitate the vectorization process:

```
void copy_v(int a[][100], int b[][100]) {
   int i,j;
   int *p_a = &a[0][0];
   int *p_b = &b[0][0];
   for (i=0; i< 3000; i+=2)
       (p_a[i], p_a[i+1]) = (p_b[i], p_b[i+1]);
}
```

The assembly output for the loop flattening example is:

```
_copy_v:
//-------------------------------------------------------------
........ procedure statistics ..........
//-------------------------------------------------------------
// Original Loop at "ExampleG.c" line 3 col 3 -- loop
// flattened into Loop at "ExampleG.c" line 4 col 5
//-------------------------------------------------------------
.................... procedure code ..................

._P1L2:
//------------------------------------------------------------
//   Loop at "ExampleG.c" line 4 col 5
//.................. loop annotations .................
//------------------------------------------------------------
//.................. loop body ...................
//------------------------------------------------------------
//   End Loop L2
//------------------------------------------------------------

._P1L3:
//------------------------------------------------------------
//   Part of top level (no loop)
//------------------------------------------------------------
// line 7
                RTS;
._copy_v.end:
```

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

Example H (Vectorization):

Consider the test program:

```
void add(short *a, short *restrict b, short *restrict c, int dim)
{
   int i, j;
   for (i = 0 ; i < dim; ++i)
       a[i] = b[i] + c[i];
}
```

for which the annotations produced are:

```
_add:
//-------------------------------------------------------------
//... procedure statistics ....
//... loop selection code ....

.P34L29:
//-------------------------------------------------------------
```

```
//   Loop at "ExampleH.c" line 3 col 3
//-------------------------------------------------------------
//   This loop executes 2 iterations of the original loop
//   in estimated 2 cycles.
//-------------------------------------------------------------
... loop body ...
//-------------------------------------------------------------
//   Loop was vectorized by a factor of 2.
//-------------------------------------------------------------
//   Vectorization peeled 1 conditional iteration from the back
//   of the loop because of an unknown trip count, possibly not a
//   multiple of 2.
//-------------------------------------------------------------
//   Consider using pragma loop_count to specify the trip count
//   or trip modulo in order to avoid conditional peeling.
//-------------------------------------------------------------
//-------------------------------------------------------------
//   End Kernel for Loop L29
//-------------------------------------------------------------
.P34L23:
//-------------------------------------------------------------
//   Loop at "ExampleH.c" line 3 col 3 (unvectorized version)
//-------------------------------------------------------------
//   This loop executes 1 iteration of the original loop in
//   estimated 2 cycles.
//-------------------------------------------------------------
//... loop body ...
//-------------------------------------------------------------
//   End Kernel for Loop L23
//-------------------------------------------------------------
//...
```

The compiler has generated two versions of the loop: a vectorized version and a non-vectorized version. The vectorized version will be executed as long as all the pointers are sufficiently aligned. The compiler has peeled a single iteration from the end of the vectorized version of the loop, which will be executed if the pointers are all aligned, but `dim` is not a multiple of two. Note that peeling could be avoided if additional information about the loop count was provided and the compiler advice "`Consider using pragma loop_count to specify the trip count or trip modulo, in order to avoid conditional peeling`" informs the user of this.

Modulo Scheduling Information

For every modulo-scheduled loop (see also "Modulo Scheduling" on page 2-79), 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 (on page 1-30) 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 Table 2-11 on page 2-90. 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 Table 2-13 on page 2-130.

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.

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 will contain 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 Table 2-13 on page 2-130 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 similar to 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,*}
...
45:     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.

Table 2-13. 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_2	I4_2, I5_2
12	epilog			I8_2	I6_2
13	epilog				I7_2
14	epilog				I8_2

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 would change 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 a source code operation results in a library call, due to lack of hardware support for performing that operation on operands of that size.
- **operation is implemented as a library call**
  This message is issued when a source code *operation* results in a library call, due to lack of direct hardware support. For instance, an integer division results in 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. In this case, the compiler must add a wrapper. 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:

- "Profiling With Instrumented Code" on page 2-135 discusses how to profile the application, measuring the time spent in individual functions in an application.
- "Stack Overflow Detection" on page 2-142 details how to use the stack overflow feature to determine when an application has exceeded its maximum stack size.

As well as providing compiler instrumented profiling, VisualDSP++ also provides statistical profiling. For more information, see "Using the Statistical Profiler" on page 2-8.

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. Compile your application with one of the `-p` switches (on page 1-65). For best results, use the optimization switches that will be enabled in the released version of the application.
2. 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 called `mon.out`.

3. Generate the profiling report, by invoking the `profblkfn` tool:

   ```
   profblkfn.exe dxe
   ```

   where *dxe* is the name of the executable.

4. Based on the profiling report, modify the application to improve performance in critical sections of code.

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.

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.

Instrumented profiling is not supported with VDK-based applications.

Generating an Application With Instrumented Profiling

The `-p` compiler switches (on page 1-65) enable 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.

To enable one of the `-p` switches in an IDDE project:

1. With the project loaded in the IDDE, select "**Project Options...**" from the "**Project**" menu.

2. Select "**Profiling**" from the "**Compile**" section in the tree pane of the "**Project Options**" dialog box.

3. Select the "**Enable compiler instrumented profiling**" check box.
4. Click the "**OK**" button.

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 either on 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 called `mon.out`, `and` will be located in the same directory as the executable.

The profiling output file needs to be converted into a readable report. This can be achieved using the command-line `profblkfn.exe` tool. See "Invoking the profblkfn.exe Command-Line Reporter" on page 2-137 for information on how to produce a report from the `mon.out` profile data file.

Invoking the profblkfn.exe Command-Line Reporter

The `profblkfn.exe` command-line tool produces a plain-text report printed to the command-line console. To produce a report:

- Invoke the `profblkfn.exe` tool (located in the top directory of your VisualDSP++ installation), providing the application executable as a parameter. For example: `profblkfn.exe test.dxe`

The report is displayed via standard output, typically to the console or command line.

Contents of the Profiling Report

The profiling report lists each profiled function called in the application, how many times it was called, and the inclusive and exclusive cycle counts for that function.

- Exclusive cycle counts include only the cycles spent processing the function. This is referred to by the "**fn only**" column in generated report files.
- Inclusive cycle counts also include the sum total of cycle counts in any function invoked from this specified function. This is referred to by the "**fn+nested**" column in generated report files.
- The cycle counts generated are the total cycles spent in all invocations of the specified function within the program.

Listing 2-2. 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 as Listing 2-2 on page 2-138, 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()`.

> 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()`.

profblkfn Command-Line Tool Report Format

The `profblkfn.exe` tool emits a report to standard output. The following is an example of the tool's output:

```
Function Name        ExecCount        Fn Only        Fn+nested
        _main                1             40               80
       _apple                3             30               30
      _banana                1             10               20
```

The "ExecCount" column contains the execution count for the given function. The "Fn Only" column contains the exclusive cycle count for a function. The "Fn+nested" column contains the inclusive cycle count for a function. For more information, see "Contents of the Profiling Report" on page 2-138.

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 will issue a diagnostic and stop storing information. The diagnostic is `'Profiler Resource Error: heap allocation failed so profiling cannot be completed'`. 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" on page 1-364.

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.

Non-Terminating Applications

When an instrumented application is executed, it records data in the application, finally flushing this data to the host computer upon termination. Non-terminating single-threaded applications are not supported, as the profiled data is never flushed to the host computer.

Profiling of Interrupts

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.

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, the profiled information will not be flushed to the host computer. To ensure the profiling information is complete, the application must terminate by unwinding its stack (returning from `main()` or their thread creation function), or by calling `exit()`.
- Unexpected flow control. Functions that perform unexpected flow control, such as 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 will attempt to complete the profiling information for any stack frames unwound, but this may be inaccurate.

Stack Overflow Detection

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`).

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.

There are many reasons why a stack overflow can occur, for example:

1. A function defines a very large local array.
2. A function defines a very large variable-length array (Refer to "Variable-Length Arrays" on page 1-166.)
3. A function uses the `alloca()` function, with an exceedingly large value as its parameter, to allocate space in the stack frame of the caller. (Refer to "System Built-In Functions" on page 1-259.)
4. The Linker Description File (`.ldf`) 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.

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.

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. This is done by either manually editing your custom Linker Description File (`.ldf`) or by regenerating your `.ldf` file once you have made the necessary adjustments to your current configuration's **Project Options: LDF Settings**.

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, function calling and other program code that leads to a stack overflow.

Compiler's Stack Overflow Detection Facility

The `-stack-detect` (on page 1-74) switch turns on the compiler's stack overflow detection facility when converting C/C++ source into assembly code. The compiler cannot generate stack overflow detection code for assembly files or files that have already been compiled to object files.

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.

3 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" on page 3-2
  provides introductory information about the ANSI/ISO standard C and C++ libraries. It also provides information about the ANSI standard header files and built-in functions that are included with this release of the `ccblkfn` compiler.

- "Documented Library Functions" on page 3-58
  tabulates the functions that are defined by ANSI standard header files.

- "C Run-Time Library Reference" on page 3-64
  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 Abridged C++ library, a conforming subset of the standard C++ library. The Abridged C++ library includes the embedded C++ and embedded standard template libraries.

This chapter describes the standard C/C++ library functions supported in the current release of the run-time libraries. Chapter 4, "DSP Run-Time Library", describes signal processing, vector, matrix, and statistical functions that assist DSP code development.

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 on the VisualDSP++ installation CD in the `Docs\Reference` folder. Viewing or printing these files requires a browser, such as Internet Explorer 6.0 (or higher). You can copy these files from the installation CD onto another disk.

C and C++ Run-Time 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" on page 3-3
- "Using the Compiler's Built-In Functions" on page 3-5
- "Linking Library Functions" on page 3-5

- "Library Attributes" on page 3-8
- "Library Function Re-Entrancy and Multi-Threaded Environments" on page 3-14
- "Working With Library Header Files" on page 3-20
- "Calling a Library Function From an ISR" on page 3-38
- "Abridged C++ Library Support" on page 3-38
- "File I/O Support" on page 3-44

For information on the C library's contents, see "C Run-Time Library Reference" on page 3-64.

For information on the Abridged C++ library's contents, see "Abridged C++ Library Support" on page 3-38.

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" on page 3-64.

Like other functions, library functions should be declared. Declarations are supplied in header files, as described in "Working With Library Header Files" on page 3-20.

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 "C/C++ and Assembly Interface" on page 1-456.

Create a VisualDSP++ project or use the archiver (`elfar`), described in the *VisualDSP++ 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. If you want to use the C/C++ run-time library functions of the same name, compile using the `-no-builtin` compiler switch (on page 1-53).

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" on page 3-64 and "DSP Run-Time Library Reference" on page 4-75.

Linking Library Functions

The C/C++ run-time library is organized as a set of run-time libraries and startup files installed under the VisualDSP++ installation directory in the `Blackfin\lib` subdirectory. Table 3-1 contains a list of these library files together with a brief description of their functions.

Table 3-1. C and C++ Library Files

Blackfin/lib Directory	Description
`crt*.doj`	C run-time startup file that sets up the system environment before calling `main()`
`crtn*.doj`	C++ cleanup file used for C++ constructors and destructors

Table 3-1. C and C++ Library Files (Cont'd)

Blackfin/lib Directory	Description
cplbtab*.doj	Default cache configuration table; memory protection and caching attributes for each Blackfin processor's memory map. See "Caching and Memory Protection" on page 1-373.
idle*.doj	Normal "termination" code that enters IDLE loop after "end" of the application
__initsbsz*.doj	Memory initializer support files
libc*.dlb	Primary ANSI C run-time library
libcpp*.dlb	Primary ANSI C++ run-time library
libcpprt*.dlb libx*.dlb	Legacy library. These libraries are empty and are provided for the sole purpose of use with a legacy .ldf file.
libdsp*.dlb	DSP run-time library
libetsi*.dlb	ETSI run-time support library
libio*.dlb	Host-based I/O facilities, as described in"stdio.h" on page 3-31
libevent*.dlb	Interrupt handler support library
libf64*.dlb	64-bit floating-point emulation routines
libprofile*.dlb	Profile support routines
librt*.dlb	C run-time support library, without file I/O
librt_fileio*.dlb	C run-time support library, with file I/O
libsftflt*.dlb	Floating-point emulation routines
libsmall*.dlb	Supervisor mode support routines
prfflg0*.doj prfflg1*.doj prfflg2*.doj	Profiling initialization routines as selected by -p, -p1, and -p2 compiler options. See "-p[1|2]" on page 1-65.

Regarding Table 3-2, in general, several versions of each C/C++ run-time library component are supplied in binary form; for example, variants are available for different Blackfin architectures while other variants have been built for use in a multi-threaded environment. Each version of a library or startup file is distinguished by a different combination of file name suffices.

Table 3-2 lists the file name suffices that may be used.

Table 3-2. C and C++ Library File Name Suffices

File Name Suffix	Description
`532`	Compiled for execution on any of the ADSP-BF522, ADSP-BF525, ADSP-BF527, ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF534, ADSP-BF536, ADSP-BF537, ADSP-BF538, ADSP-BF539, ADSP-BF542, ADSP-BF544, ADSP-BF548, or ADSP-BF549 processors
`535`	Compiled for execution on any of the ADSP-BF535 processors
`561`	Compiled for execution on the ADSP-BF561 processors
`a`	Compiled for execution on core A of a dual-core processor
`b`	Compiled for execution on core B of a dual-core processor
`mt`	Built for multi-thread environments
`x`	Libraries compiled with C++ exception handling enabled
`y`	Compiled with the `-si-revision` switch (on page 1-74) to avoid all known hardware anomalies

As an example, the C run-time library `libc535mty.dlb` has been compiled with the `-si-revision` switch (on page 1-74) for execution on any ADSP-BF535 processor, and has been built for VDK multi-threaded environments.

Code or data built to run on a specific processor rather than a group of processors described in Table 3-2 has a file name suffix indicating the target part. For example, `cplbtab531.doj` contains the default cache configuration for the ADSP-BF531 only.

The C/C++ run-time library provides further variants of the start-up files (`crt*.doj`) that have been built from a single source file. (See "Startup

Code Overview" on page 1-357.) Table 3-3 shows the file name suffices used to differentiate between different versions of this binary file.

Table 3-3. 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`
`p`	Startup file used by applications that have been compiled with profiling instrumentation
`s`	Startup file used by applications that run in supervisor mode

For example, `crtcf535.doj` is the start-up file that enables file I/O support and initializes a C++ application that has been compiled to execute in user mode on an ADSP-BF535 processor.

When an application calls a C/C++ library function, the call creates a reference that the linker resolves. One way to direct the linker to the location of the appropriate run-time library is to use the default linker description file (<*your_target*>`.ldf`). If you are using a customized `.ldf` file to link the application, add the appropriate library/libraries and startup files to the `.ldf` file used by the project.

Instead of modifying a customized `.ldf` file, use the compiler's `-l` switch to specify the names of libraries to be searched by the linker. For example, the switches `-lc532 -lcpp532` add the C library `libc532.dlb` and the C++ library `libcpp532.dlb` to the list of libraries that the linker examines. For more information on the `.ldf` file, refer to the *VisualDSP++ Linker and Utilities Manual*.

Library Attributes

The run-time libraries make use of file attributes. (See "File Attributes" on page 1-471 for details on using file attributes.)

All object files within the run-time libraries listed in Table 3-1 on page 3-5 have the attributes listed in Table 3-4.

For each object (obj) in the run-time libraries, the following is true:

Table 3-4. Run-Time Library Object Attributes

Attribute Name	Meaning of Attribute and Value
libGroup	A potentially multi-valued attribute. Each value is the name of a header file that either defines obj or 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 libdsp532y.dlb, then the value of the attribute would be libdsp.
libFunc	The name of all the functions in obj. libFunc will have multiple values – both the C and assembly linkage names will be listed. libFunc will also contain all the published C and assembly linkage names of objects in obj's library that call into obj.
prefersMem	One of three values: internal, external, or any. If obj contains a function that is likely to be application performance-critical, it will be marked as internal. Most DSP run-time library functions fit into the internal category. If a function is deemed unlikely to be essential for achieving the necessary performance, it will be marked as external (all 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`

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

File: `func2.c`

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

File: `subfunc.c`

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

The objects in `libfunc.dlb` will have the attributes as defined in Table 3-5.

Table 3-5. Attribute Values in `libfunc.dlb`

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

Table 3-5. Attribute Values in `libfunc.dlb` (Cont'd)

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

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

Exceptions to Library Attribute Conventions

This section lists exceptions to the library attribute conventions.

The C++ support libraries (`libcpp*.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, the return type, and the 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.

Table 3-6 lists additional `libGroup` attribute values.

Table 3-6. Additional `libGroup` Attributes

Value	Meaning
`exceptions_support`	Compiler support routines for C++ exceptions
`floating_point_support`	Compiler support routines for floating-point arithmetic
`integer_support`	Compiler support routines for integer arithmetic
`runtime_support`	Other run-time functions that do not fit into any of the above categories

Objects with any of the `libGroup` attribute values listed in Table 3-6 will not contain the `libGroup` or `libFunc` attributes from any calling objects.

Table 3-7 summarizes the default memory placement using `prefersMem`.

Table 3-7. Default Memory Placement Summary

Library	Placement

`__initsbsz*.doj`
`crt*.doj`
`crtn*.doj`
`cplbtab*.doj`
`mc_data561*.doj`
	Hard placement using sections

`libcpp*.dlb` `libetsi*.dlb`	Any (`any`)

`idle*.doj`
`libio*.dlb`
`libprofile*.dlb`
`prfflg*.doj`
	External (`external`)

`libf64ieee*.dlb` `libsftflt*.dlb`	Internal (`internal`)
`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`

Table 3-7. Default Memory Placement Summary (Cont'd)

`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
`libmc561*.dlb`	`any` apart from `exit`, which is `external`
`librt*.dlb`	`internal` for `_memcpy` and `_memmove`, otherwise `any`
`libsmall*.dlb`	`any` or `external`, except for the vector table for `signal` and `interrupt`, which is `internal`

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

Mapping Objects to Flash 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 *VisualDSP++ Linker and Utilities Manual* for information on memory initialization.

Library Function Re-Entrancy and Multi-Threaded Environments

The C/C++ run-time libraries are not re-entrant. For example, it is not possible to put the library code into L2 memory on the ADSP-BF561 processor and have either core (core A or core B) call the libraries without using a user-defined semaphore.

It is sometimes desirable to have several active instances of a given library function at once. This can occur because:

- An interrupt or other external event invokes a function, while the application is also executing that function.
- The application uses a multi-threaded architecture, such as VDK, and more than one thread executes the function concurrently.
- The application is built for a multi-core processor, such as the ADSP-BF561 processor, and more than one core is executing the function concurrently.

When multiple concurrent threads may be active at once, ensure that the library functions used are able to support this activity. If a function uses private data storage, and both active instances of the function modify the same storage area without due care, undefined behavior may occur.

The majority of the C/C++ run-time library functions are safe in this regard, in that the functions do not have private storage, operating instead on parameters passed by the caller.

A small subset of library functions use private storage, and functions like the `stdio` support operate on shared resources (like `FILE` pointers) that must be safely updated. To support these needs, multi-threaded builds of the run-time libraries are included in VisualDSP++.

The multi-threaded versions of the run-time libraries use local storage routines for thread-local and core-local private copies of data. (See "adi_obtain_mc_slot, adi_free_mc_slot, adi_set_mc_value, adi_get_mc_value" on page 3-76.) Recursive locking mechanisms are included so that shared resources, such as `stdio FILE` buffers, are only updated by a single function instance at any given time.

The differences between the multi-threaded libraries and the multi-core libraries are:

- Multi-threaded libraries have "`mt`" in their file name, and are built for an arbitrary number of concurrent threads, as is the case for VDK applications. Multi-threaded libraries are used both for VDK builds and for non-VDK builds on dual-core processors, when the `-threads` compiler switch (on page 1-76) is specified.
- Multi-core libraries have "`mc`" in their file name, and are built for dual-core applications, with a single thread running on each of two cores in a dual-core processor. Multi-core libraries are used for non-VDK builds on dual-core processors, when the `-multicore` compiler switch (on page 1-50) is specified.

The following Standard library elements use thread-local or core-local private storage:

- `atexit` function
- `rand` function
- `strtok` function
- `asctime` function
- `errno` global variable

The `atexit` function requires a core-local slot, but not a thread-local slot, because VDK applications do not use `exit` to terminate each thread, and effectively run "forever". Using `exit` terminates the whole VDK application. By contrast, in a multi-core application, `exit` terminates the application in one core while the other continues.

You must specify the `-multicore` compiler switch when building multi-core applications. The switch has the following effects:

- At compile-time, it defines the `__ADI_MULTICORE` macro to ensure that core-local storage operations are available.
- At link-time, it ensures that the application is linked against the multi-threaded and multi-core builds of the library.
- It repositions the default heap to be in shared memory. Allocations by either core will be served by the same heap. The heap allocation and release routines use locking to ensure that only one core at a time is updating the heap resource records.

While thread-safe variants of the C/C++ run-time libraries exist, many functions are not interrupt-safe as they access global data structures. It is therefore recommended that ISRs do not call library functions, as unexpected behavior may result if the interrupt occurs during a call to such a function.

An alternative approach is to disable interrupts before the application makes run-time library calls. This may be disadvantageous for time-critical applications as interrupts may be disabled for a long period of time. The DSP run-time library functions do not modify global data structures and are therefore interrupt-safe.

Support Functions for Private Data

The run-time library provides support functions for creating thread-local and core-local private data storage.

For thread-local private storage, refer to the *VisualDSP++ Kernel (VDK) User's Guide*.

For core-local private storage, see "adi_obtain_mc_slot, adi_free_mc_slot, adi_set_mc_value, adi_get_mc_value" on page 3-76.

Support Functions for Locking

The run-time library provides support functions in the form of locking routines to ensure safe access to shared resources. See "adi_acquire_lock, adi_try_lock, adi_release_lock" on page 3-71 for more information.

Other Support Functions for Multi-Core Applications

The run-time library includes the `adi_core_id` function, which can be used by shared code to determine which core is executing it. See "adi_core_id" on page 3-74 for more details.

Library Placement

A multi-threaded or multi-core application has some storage that must be shared across threads and cores (such as locks, that must be globally accessible), and some storage that must be private (such as the C++ exception look-up tables in a multi-core application). Table 3-8 lists requirements for each of the libraries, regarding section placement.

Table 3-8. Object/Library Multi-Core Restrictions

Object/Library	Restriction

`__init*.doj`
`cplbtab*.doj`
`crt*.doj`
	The startup and configuration objects are core-specific, and must not be placed in a shared memory section

`libc*.dlb`	Can be placed in a shared memory section
`libcpp*.dlb`	Cannot be placed in a shared memory section
`libdsp*.dlb`	Can be placed in a shared memory section
`libetsi*.dlb`	Can be placed in a shared memory section, as provided. If rebuilt in debug mode, so that Overflow and Carry "flags" are maintained, these global variables will not be locked during updates.
`libevent*.dlb`	Single-core memory placement recommended
`libmc*.dlb`	Can be placed in a shared memory section
`mc_data*.doj`	Must be placed in a shared memory section

Table 3-8. Object/Library Multi-Core Restrictions (Cont'd)

Object/Library	Restriction
libprofile*.dlb	Not supported in a multi-core environment
librt*.dlb	Can be placed in a shared memory section
librt_fileio.dlb	Can be placed in a shared memory section
libsftflt*.dlb	Can be placed in a shared memory section
libsmall*.dlb	Can be placed in a shared memory section
libx*.dlb	Cannot be placed in a shared memory section
prfflg*.doj	Not supported in a dual-core environment

Section Placement

Libraries are mapped into shared or private areas via sections within the .ldf file. Table 3-9 shows which LDF output sections must be shared, and which must be private.

Table 3-9. Shared and Private LDF Output Sections

LDF Section	Must Be Shared	Must Be Core-Specific
primio_atomic_lock	Yes	No
mc_data	Yes	No
heap	No	No
L1_code	No	Yes
cplb	No	Yes
cplb_data	No	Yes
bsz	No	Yes
bsz_init	No	Yes
.edt	No	Yes
.cht	No	Yes
constdata	No	Yes

Table 3-9. Shared and Private LDF Output Sections (Cont'd)

LDF Section	Must Be Shared	Must Be Core-Specific
`ctor`	No	Yes
`ctorl`	No	Yes
`.gdt`	No	Yes
`.gdtl`	No	Yes
`.frt`	No	Yes
`.frtl`	No	Yes
`stack`	No	Yes

Note that the sharing or privacy of the "`heap`" section is a matter for the application designer. A multi-core application defaults to using a shared heap with appropriate locking during allocation and release, but a private per-core heap may better suit the application.

Any sections not listed in Table 3-9 may be shared or private, according to the discretion of the application developer. Ensure that shared sections use appropriate locking mechanisms to avoid corruption by simultaneous accesses.

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.

Table 3-10 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" on page 1-3) to augment information supplied in this chapter.

Table 3-10. Standard C Run-Time Library Header Files

Header	Purpose	Standard
aditypes.h	Type definitions (on page 3-22)	Analog extension
assert.h	Diagnostics (on page 3-22)	ANSI
ccblkfn.h	Access to system facilities on Blackfin processors (on page 3-23)	Analog extension
cplbtab.h	Support routines for cache-related setup and management routines (on page 3-23)	Analog extension
ctype.h	Character handling (on page 3-23)	ANSI
device.h	Macros and data structures for alternative device drivers (on page 3-24)	Analog extension
device_int.h	Enumerations and prototypes for alternative device drivers (on page 3-24)	Analog extension
errno.h	Error handling (on page 3-24)	ANSI
float.h	Floating point (on page 3-24)	ANSI
iso646.h	Boolean operators (on page 3-25)	ANSI
limits.h	Limits (on page 3-26)	ANSI
locale.h	Localization (on page 3-26)	ANSI
math.h	Mathematics (on page 3-26)	ANSI
mc_data.h	Routines for accessing the core-specific data for multi-core processors (on page 3-28)	Analog extension
misra_types.h	Exact-width integer types (on page 3-28)	MISRA-C:2004
setjmp.h	Non-local jumps (on page 3-28)	ANSI
signal.h	Signal handling (on page 3-28)	ANSI
stdarg.h	Variable arguments (on page 3-28)	ANSI
stdbool.h	Boolean macros (on page 3-29)	ANSI
stddef.h	Standard definitions (on page 3-29)	ANSI
stdfix.h	Fixed point (on page 3-29)	ISO/IEC TR 18037
stdint.h	Exact width integer types (on page 3-29)	ANSI

Table 3-10. Standard C Run-Time Library Header Files (Cont'd)

Header	Purpose	Standard
stdio.h	Input/output (on page 3-31)	ANSI
stdlib.h	Standard library (on page 3-36)	ANSI
string.h	String handling (on page 3-36)	ANSI
time.h	Date and time (on page 3-36)	ANSI

The following sections describe the header files contained in the C library. The header files are listed in alphabetical order.

adi_types.h

The adi_types.h header file contains the type definitions for char_t, float32_t, and float64_t. The adi_types.h header file also includes stdint.h (on page 3-29) and stdbool.h (on page 3-29).

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 (on page 3-65) to terminate the application. The message displayed by the assert macro has the following form:

filename : linenumber expression – run-time 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.

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 contains the asserts. For example:

```
#pragma default_section(STRINGS,"sdram_bank1")
```

This will move all strings—not just those associated with assert.

Alternatively, place the `-section` flag on the compiler command line or include the option via **Project Options**-> **Compile**-> **General**->**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 Table 3-18 on page 3-59).

cplbtab.h

The `cplbtab.h` header file (see Table 3-19 on page 3-59) provides support routines for cache-related setup and management routines, such as `enable_data_cache()` and `cplb_init()`.

ctype.h

The `ctype.h` header file (see Table 3-20 on page 3-59) contains functions for character handling, such as `isalpha`, `tolower`, and so on.

device.h

The `device.h` header file provides macros and defines data structures required by an alternative device driver to provide file input and output services for `stdio` library functions. Normally, `stdio` functions use a default driver to access an underlying device, but alternative device drivers may be registered that may then be used transparently by these functions. This mechanism is described in "Extending I/O Support to New Devices" on page 3-44.

device_int.h

The `device_int.h` header file contains function prototypes and provides enumerations for alterative device drivers. An alternative device driver is normally provided by an application and may be used by the `stdio` library functions to access an underlying device; an alternative device driver may coexist with, or may replace, the default driver that is supported by the VisualDSP++ simulator and EZ-KIT Lite® evaluation systems. Refer to "Extending I/O Support to New Devices" on page 3-44 for information.

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}" on page 1-34.)

iso646.h

The iso646.h header file defines symbolic names for certain C (Boolean) operators. Table 3-11 shows symbolic names and their associated value.

Table 3-11. Symbolic Names Defined in iso646.h

Symbolic Name	Equivalent
and	&&
and_eq	&=
bitand	&
bitor	|
compl	~

Table 3-11. Symbolic Names Defined in iso646.h (Cont'd)

Symbolic Name	Equivalent
`not`	`!`
`not_eq`	`!=`
`or`	`||`
`or_eq`	`|=`
`xor`	`^`
`xor_eq`	`^=`

The symbolic names have the same name as the C++ keywords that are accepted by the compiler when the `-alttok` switch is specified. (For more information, see "-alttok" on page 1-28.)

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 Table 3-21 on page 3-60) 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 fractional data and 32-bit fractional data.

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 (on page 1-34) 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 un-suffixed 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 infinity.

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.

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 Table 3-22 on page 3-60) 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" on page 3-29 and "stdbool.h" on page 3-29, plus data types char_t, float32_t, and float64_t.

setjmp.h

The setjmp.h header file (see Table 3-23 on page 3-60) contains setjmp and longjmp for non-local jumps.

signal.h

The signal.h header file (see Table 3-24 on page 3-61) provides function prototypes for the standard ANSI signal.h routines. The signal handling functions process conditions (hardware signals) that may occur during program execution. They determine the way that C programs respond to these signals. These functions are designed to process signals such as external interrupts and timer interrupts.

stdarg.h

The stdarg.h header file (see Table 3-25 on page 3-61) 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 three Boolean-related macros (`true`, `false`, and `__bool_true_false_are_defined`) and an associated data type (`bool`). The `stdbool.h` header file was introduced in the C99 standard library.

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" on page 1-104.

stddef.h

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

stdint.h

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

Table 3-12 show each of the `typedef`s defined by the header file, and documents the macro name of the associated minimum and maximum values for the types.

Table 3-12. Exact-Width Integer Types

Type	Common Equivalent	MIN	MAX
`int8_t`	`signed char`	`INT8_MIN`	`INT8_MAX`
`int16_t`	`short`	`INT16_MIN`	`INT16_MAX`

Table 3-12. Exact-Width Integer Types (Cont'd)

Type	Common Equivalent	MIN	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

Table 3-13 describes MIN and MAX macros defined for `typedefs` in other headings.

Table 3-13. 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 Table 3-27 on page 3-61) 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-64` switch (on page 1-34). 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 (on page 1-40), 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. This library is fully compatible with previous releases of VisualDSP++ (version 4.5 and earlier). 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 VisualDSP++ 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 (on page 1-38). 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 VisualDSP++ 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 alternative libraries have different conventions for printing IEEE floating-point values that are either NaN's (Not-A-Number) or Infinity. The third-party I/O library also accepts `nan` and `inf` (in any case) as input for the `e`, `f`, and `g` conversion specifiers.
- 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 with a device driver that provides a set of low-level primitives for `open`, `close`, `read`, `write`, and `seek` operations. By default, these operations are provided by the VisualDSP++ simulator and EZ-KIT Lite systems; this mechanism is outlined in "Default Device Driver Interface" on page 3-53. However, alternative device drivers may be registered (see "Extending I/O Support to New Devices" on page 3-44) that can then be used transparently through the `stdio.h` functions.

Applications should be aware that the default device driver 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 repositioning 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 default device driver will disable interrupts, and will halt the DSP while it negotiates with the host to perform the required I/O operation. Once the I/O operation has completed, the default device driver will restart the DSP and then re-enable interrupts.

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

The following restrictions apply to either library in this software release:

- Functions `rename()` and `remove()` are only supported under the default device driver supplied by the VisualDSP++ simulator and EZ-KIT Lite system, and only operate on the host file system.
- Positioning within a file that has been opened as a text stream is only supported if the lines within the file are terminated by the character sequence `\r\n`.
- Support for formatted reading and writing of data of 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 Table 3-28 on page 3-62) 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 Table 3-29 on page 3-63) contains string handling functions, including `strcpy` and `memcpy`.

time.h

The `time.h` header file (see Table 3-30 on page 3-63) 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            */
};
```

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.

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=600000LL`
- Via the System Services Library
- Via the **Processor speed** option, specified in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Processor (1)** category

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. Additionally, the memory allocation routines (such as `malloc`, `calloc`, `realloc`, and `free`) and the C++ operators (`new` and `delete`) read and update global tables and are not interrupt-safe; they should not be called from an ISR.

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 VDK multi-threaded environment or by dual-core Blackfin applications. These library functions are found in the run-time libraries that have an `_mt` suffix in their file names.

Abridged C++ Library Support

In C++ mode, the compiler can call many functions from the Abridged C++ library, which is a conforming subset of the C++ library.

The Abridged C++ library has two major components: the embedded C++ library (EC++), and the embedded standard template library (ESTL). The embedded C++ library is a conforming implementation of the embedded C++ library as specified by the Embedded C++ Technical Committee. You can view the Abridged Library Reference by locating the file <install_path>`\Docs\cpl_lib\index.html` and opening it in a Web browser.

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

- "Embedded C++ Library Header Files" on page 3-39
- "C++ Header Files for C Library Facilities" on page 3-41
- "Embedded Standard Template Library (ESTL) Header Files" on page 3-42
- "Using Thread-Safe C/C++ Run-Time Libraries With VDK" on page 3-43

Embedded C++ Library Header Files

Table 3-14 describes the header files in the embedded C++ library.

Table 3-14. 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`. This implementation does not support the full set of complex operations as specified by the C++ standard. In particular, it does not support either the transcendental functions or the I/O operators "<<" and ">>". The `complex` header file and the C library header file `complex.h` refer to two different and incompatible implementations of the `complex` data type.
`exception`	Defines the `exception` and `bad_exception` classes and several functions for low-level exception handling. These functions are used as the basis for higher-level exception handling in `<stdexcept>` (See "stdexcept" below.)

Table 3-14. Embedded C++ Library Header Files (Cont'd)

Header	Description
`fract`	Defines the `fract` data type, which supports fractional arithmetic, assignment, and type-conversion operations using a 32-bit data type. The header file is fully described under "Fractional Value Built-In Functions in C++" on page 1-232.
`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` template 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`.
`new`	Declares several classes and functions for memory allocations and deallocations.
`ostream`	Defines the `ostream` class for `iostream` insertions.
`shortfract`	Defines the `shortfract` fractional class, which supports fractional arithmetic, assignment, and type-conversion operations using a 16-bit base type. The header file is fully described under "Fractional Value Built-In Functions in C++" on page 1-232.
`sstream`	Defines the `stringbuf`, `istringstream`, and `ostringstream` classes for various `string` object manipulations.
`stdexcept`	Defines a variety of classes for exception reporting.
`streambuf`	Defines the `streambuf` classes for basic operations of the `iostream` classes. Note that most of the `iostream` header files include `streambuf`.

Table 3-14. Embedded C++ Library Header Files (Cont'd)

Header	Description
`string`	Defines the `string` template and various supporting classes and functions for `string` manipulations. Objects of the `string` type should not be confused with the null-terminated C strings.
`strstream`	Defines the `strstreambuf`, `istrstream`, and `ostream` classes for `iostream` manipulations on allocated, extended, and freed character sequences.

C++ Header Files for C Library Facilities

For each C standard library header, there is a corresponding standard C++ header. 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`.

Table 3-15 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 3-15. 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

Table 3-15. C++ Header Files for C Library Facilities (Cont'd)

Header	Description
csetjmp	Executes non-local goto statements
csignal	Controls various exceptional conditions
cstdarg	Accesses a variable number of arguments
cstddef	Defines several useful data types and macros
cstdio	Performs input and output
cstdlib	Performs a variety of operations
cstring	Manipulates several kinds of strings

Embedded Standard Template Library (ESTL) Header Files

Templates and the associated header files are not part of the embedded C++ standard library, but are supported by the compiler in C++ mode. Table 3-16 describes embedded standard template library header files.

Table 3-16. Embedded Standard Template Library (ESTL) 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

Table 3-16. Embedded Standard Template Library (ESTL) Header Files (Cont'd)

Header	Description
`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

The Embedded C++ library also includes several headers for compatibility with traditional C++ libraries; see Table 3-17.

Table 3-17. Embedded Standard Template 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

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

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

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

File I/O Support

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

This section describes:

- "Extending I/O Support to New Devices" on page 3-44
- "Default Device Driver Interface" on page 3-53

Refer to "stdio.h" on page 3-31 for information about the conventional C input and output facilities provided by the compiler.

Extending I/O Support to New Devices

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

This section describes:

- "DevEntry Structure" on page 3-45
- "Registering New Devices" on page 3-50
- "Pre-Registering Devices" on page 3-50
- "Default Device" on page 3-52
- "Remove and Rename Functions" on page 3-53

DevEntry Structure

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

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

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

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

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

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

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

init:
The `init` field is a pointer to an initialization function. The run-time library calls this function when the device is first registered, passing in the address of this structure (and thus giving the `init` function access to

`DeviceID` and the field `data`). If the `init` function encounters an error, it must return -1; otherwise, it returns a positive value to indicate success.

open:
The `open` field is a pointer to a function that performs the "*`open file`*" operation upon the device. The run-time library calls this function in response to requests such as `fopen()`, when the device is the currently selected default device. The `name` parameter is the path name to the file to be opened, and the `mode` parameter is a bitmask that indicates how the file is to be opened. Bits 0-1 indicate reading and/or writing.

```
0x0000   Open file for reading
0x0001   Open file for writing
0x0002   Open file for reading and writing
0x0003   Invalid
```

Additional bits may be OR'd into `mode` to alter the file's behavior, such as:

```
0x0008   Open the file for appending
0x0100   Create the file, if it does not already exist.
0x0200   Truncate the file to zero length, if it
         already exists
0x4000   Open the file as a text stream (converting the
         character sequence \r\n to \n on reading,
         and \n to \r\n on writing).
0x8000   Open the file as a binary stream (raw mode).
```

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

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

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

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

The `write` function takes three parameters:

- `fd` – This file descriptor identifies the file to be written to. It will be a value returned from a previous call to the open function.
- `buf` – A pointer to the data to be written to the file
- `size` – The number of bytes to be written to the file

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

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

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

The `read` function's parameters are:

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

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

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

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

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

The `seek` function takes the following parameters:

- `fd` – The file descriptor for the file which will have its read/write position altered
- `offset` – A value used to determine the new read/write pointer position within the file; it is in 8-bit bytes

- `whence` – A value that indicates how the `offset` parameter is interpreted:
  - 0: `offset` is an absolute value, giving the new read/write position in the file
  - 1: `offset` is a value relative to the current position within the file
  - 2: `offset` is a value relative to the end of the file

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

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

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

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

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

Registering New Devices

A new device can be registered with the following function:

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

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

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

- The `DeviceID` is the same as another, already registered device
- There are no more slots left in the device registry table
- The `DeviceID` is less than zero
- Some of the function pointers are NULL
- The device's `init()` routine returned a failure result
- The device attempted to claim a standard stream that is already claimed by another device

Pre-Registering Devices

The library source file, `devtab.c`, which is located under a VisualDSP++ installation in the `Blackfin/lib/src/libio` subdirectory, declares the array:

```
DevEntry_t DevDrvTable[];
```

This array contains pointers to `DevEntry` structures for each pre-registered device, (that is, devices that are available as soon as `main()` is entered), and that do not need to be registered at runtime by calling `add_devtab_entry()`.

By default, the "*PrimIO*" device is registered. The `PrimIO` device provides support for target/host communication when using simulators and the Analog Devices emulators and debug agents. This device is pre-registered, so that `printf()` and similar functions operate as expected without additional setup.

Pre-register additional devices, as follows:

1. Add a copy of the `devtab.c` source file to your project.

2. Declare your new device's `DevEntry` structure within the `devtab.c` file, for example,

   ```
   extern DevEntry myDevice;
   ```

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

   ```
   DevEntry_t DevDrvTable[MAXDEV] = {
   #ifdef PRIMIO
      &primio_deventry,
   #endif
      &myDevice,   /* new pre-registered device */
      0,
   };
   ```

Pre-registered devices are initialized automatically when device I/O is first used. The run-time library calls the `init()` function of each of the pre-registered devices in turn.

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

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

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

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

Default Device

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

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

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

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

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

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

Remove and Rename Functions

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

Default Device Driver Interface

The `stdio` functions provide access to the files on a host system through a device driver that supports a set of low-level I/O primitives, which are described under "Extending I/O Support to New Devices" on page 3-44. The default device driver implements these primitives based on a simple interface provided by the VisualDSP++ simulator and EZ-KIT Lite evaluation systems.

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

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

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

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

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

Data Packing for Primitive I/O

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

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

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

Data Structure for Primitive I/O

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

```
typedef struct
{
   enum
   {
      PRIM_OPEN = 100,
      PRIM_READ,
      PRIM_WRITE,
      PRIM_CLOSE,
      PRIM_SEEK,
      PRIM_REMOVE,
      PRIM_RENAME
   } op;
   int   fileID;
   int   flags;
   unsigned char *buf;   /* data buffer, or file name     */
   int   nDesired;       /* number of characters to read  */
                         /* or write                      */
   int   nCompleted;     /* number of characters actually */
                         /* read or written               */
   void  *more;          /* for future use                */
}
PrimIOCB_T;
```

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

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

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

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

```
M_OPENR = 0x0001      /* open for reading                  */
M_OPENW = 0x0002      /* open for writing                  */
M_OPENA = 0x0004      /* open for append                   */
M_TRUNCATE = 0x0008   /* truncate to zero length           */
                      /* if file exists                    */
M_CREATE = 0x0010     /* create the file if necessary      */
M_BINARY = 0x0020     /* binary file (vs. text file)       */
M_STRLEN_PROVIDED = 0x8000  /* length of file name(s) avail. */
```

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

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

```
M_ALIGN_BUFFER = 0x10
```

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

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

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

```
enum
{
   M_SEEK_SET = 0x0001,   /* seek origin is the start of
                             the file                  */
   M_SEEK_CUR = 0x0002,   /* seek origin is the current
                             position within the file  */
   M_SEEK_END = 0x0004,   /* seek origin is the end of
                             the file                  */
};
```

The flags field is unused for a CLOSE request.

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

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

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

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

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

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

Documented Library Functions

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

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

Table 3-18 lists functions in the `ccblkfn.h` header file. For more information, see "ccblkfn.h" on page 3-23.

Table 3-18. Library Functions in the `ccblkfn.h` Header File

adi_obtain_mc_slot, adi_free_mc_slot, adi_set_mc_value, adi_get_mc_value	adi_core_id	_l1_memcpy, _memcpy_l1

Table 3-19 lists functions in the `cplbtab.h` header file. For more information, see "cplbtab.h" on page 3-23.

Table 3-19. Library Functions in the `cplbtab.h` Header File

cplb_hdr	cache_invalidate	cplb_mgr
disable_data_cache	enable_data_cache	flush_data_cache
cplb_init		

Table 3-20 lists functions in the `ctype.h` header file. For more information, see "ctype.h" on page 3-23.

Table 3-20. Library Functions in the `ctype.h` Header File

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

Table 3-21 lists functions in the `math.h` header file. For more information, see "math.h" on page 3-26.

Table 3-21. Library Functions in the `math.h` Header File

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

Table 3-22 lists functions in the `mc_data.h` header file. For more information, see "mc_data.h" on page 3-28.

Table 3-22. Library Functions in the `mc_data.h` Header File

adi_obtain_mc_slot, adi_free_mc_slot, adi_set_mc_value, adi_get_mc_value

Table 3-23 lists functions in the `setjmp.h` header file. For more information, see "setjmp.h" on page 3-28.

Table 3-23. Library Functions in the `setjmp.h` Header File

longjmp	setjmp

Table 3-24 lists functions in the `signal.h` header file. For more information, see "signal.h" on page 3-28.

Table 3-24. Library Functions in the `signal.h` Header File

raise	signal	interrupt

Table 3-25 lists functions in the `stdarg.h` header file. For more information, see "stdarg.h" on page 3-28.

Table 3-25. Library Functions in the `stdarg.h` Header File

va_arg	va_end	va_start

Table 3-26 lists functions in the `stdfix.h` header file. For more information, see "stdfix.h" on page 3-29.

Table 3-26. Library Functions in the `stdfix.h` Header File

absfx	bitsfx	countlsfx
divifx	fxbits	fxdivi
idivfx	mulifx	roundfx
strtofxfx		

Table 3-27 lists functions in the `stdio.h` header file. For more information, see "stdio.h" on page 3-31.

Table 3-27. Supported Library Functions in the `stdio.h` Header File

clearerr	fclose	feof
ferror	fflush	fgetc
fgetpos	fgets	fprintf
fputc	fputs	fopen

Table 3-27. Supported Library Functions in the `stdio.h` Header File (Cont'd)

freopen	fscanf	fread
fseek	fsetpos	ftell
fwrite	getc	getchar
gets	perror	printf
putc	putchar	puts
remove	rename	rewind
scanf	setbuf	setvbuf
snprintf	sprintf	sscanf
tmpfile	tmpnam	ungetc
vfprintf	vprintf	vsnprintf
vsprintf		

Table 3-28 lists functions in the `stdlib.h` header file. For more information, see "stdlib.h" on page 3-36.

Table 3-28. Library Functions in `stdlib.h` Header File

abort	abs	atexit
atof	atoi	atol
	atoll	bsearch
calloc	div	exit
free	heap_calloc	heap_free
heap_init	heap_install	heap_lookup
heap_malloc	heap_realloc	heap_space_unused
labs	ldiv	malloc
qsort	rand	realloc
space_unused	srand	strtod

Table 3-28. Library Functions in `stdlib.h` Header File (Cont'd)

strtof	strtol	strtold
strtoll	strtoul	strtoull

Table 3-29 lists functions in the `string.h` header file. For more information, see "string.h" on page 3-36.

Table 3-29. Library Functions in `string.h` Header File

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

Table 3-30 lists functions in the `time.h` header file. For more information, see "time.h" on page 3-36.

Table 3-30. Library Functions in `time.h` Header File

asctime	clock	ctime
difftime	gmtime	localtime
mktime	strftime	time

C Run-Time Library Reference

The C run-time library is a collection of functions 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** and Purpose of the function
- **Synopsis** – Required header file and functional prototype
- **Description** – Function specification
- **Error Conditions** – Method that the functions use to indicate an error
- **Example** – Typical function usage
- **See Also** – Related functions

abort

Abnormal program end

Synopsis

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

Description

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

Error Conditions

The `abort` function does not return.

Example

```
#include <stdlib.h>
extern int errors;

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

See Also

atexit, exit

abs

Absolute value

Synopsis

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

Description

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

Note: The result of `abs(INT_MIN)` is undefined.

Error Conditions

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

Example

```
#include <stdlib.h>
int i;
i = abs(-5);        /* i == 5 */
```

See Also

absfx, fabs, labs

absfx

absolute value

Synopsis

```
#include <stdfix.h>

fract absr(fract f);
accum absk(accum a);

short fract abshr(short fract f);
short accum abshk(short accum a);

long fract abslr(long fract f);
long accum abslk(long accum a);
```

Description

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

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

Error Conditions

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

Example

```
#include <stdfix.h>
accum a;
long fract f;
a = abshk(-12.5k);          /* a == 12.5k */
f = abslr(0.75lr);          /* f == 0.75lr */

#if defined(_C99)
a = absfx(-12.5k);          /* a == 12.5k */
f = absfx(0.75lr);          /* f == 0.75lr */
#endif
```

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 the function results are in radians.

The input for the functions `acos`, `acosf`, and `acosd` must be in the range [-1, 1], and the functions return a result that will be 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`].

Error Conditions

The arc cosine functions return a zero if the input is not in the defined range.

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.

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);
}
```

> 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 someting 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_id

Identify caller's core

Synopsis

```
#include <ccblkfn.h>
int adi_core_id(void);
```

Description

The `adi_core_id` function returns a numeric 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 a zero value when executed by core A, and a value of one when executed on core B.

Error Conditions

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

Example

```
#include <ccblkfn.h>

const char *core_name(void)
{
      if (adi_core_id() == 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

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, then allocate a slot.
2. If this is the first time this core has needed the private data, then allocate storage for the data and record it in the slot, else 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 will return the slot's identifier, via the `slotID` pointer, and will return 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_get_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_get_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;
}
```

The multi-core private storage routines do not disable interrupts; that is left at the caller's discretion.

asctime

Convert broken-down time into a string

Synopsis

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

Description

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

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

where

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

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

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

Error Conditions

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

Example

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

struct tm tm_date;

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

See Also

ctime, gmtime, localtime

asin

Arc sine

Synopsis

```
#include <math.h>

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

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`].

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 [$-\pi/2$, $\pi/2$].

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 [$-\pi/4$, $\pi/4$].

Error Conditions

The arc tangent functions do not return an error condition.

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).

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.

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.

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_r_t bitsr(fract f);
int_k_t bitsk(accum a);

int_hr_t bitshr(short fract f);
int_hk_t bitshk(short accum a);

int_lr_t bitslr(long fract f);
int_lk_t bitslk(long accum a);

uint_ur_t bitsur(unsigned fract f);
uint_uk_t bitsuk(unsigned accum a);

uint_uhr_t bitsuhr(unsigned short fract f);
uint_uhk_t bitsuhk(unsigned short accum a);

uint_ulr_t bitsulr(unsigned long fract f);
uint_ulk_t bitsulk(unsigned long accum a);
```

Description

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

Error Conditions

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

Example

```
#include <stdfix.h>
#include <fract.h>

int_k_t k;
uint_ulr_t ulr;

fract16 fr16;
fract32 fr32;

k = bitsk(-12.5k);            /* k == 0xfffffff9c0000000 */
ulr = bitsulr(0.125ulr);      /* ulr == 0x20000000 */

fr16 = bitsr (-0.75r);        /* fr16 = 0x6000 */
fr32 = bitslr (0.25lr);       /* fr32 = 0xe0000000 */
```

See Also

fxbits

bsearch

Perform binary search in a sorted array

Synopsis

```
#include <stdlib.h>

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

Description

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

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

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

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

Error Conditions

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

Example

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

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

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

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

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

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

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

See Also

qsort

cache_invalidate

Invalidate processor instruction and data caches

Synopsis

```
#include <cplbtab.h>

void cache_invalidate(int cachemask);
void icache_invalidate(void);
void dcache_invalidate(int a_or_b);
void dcache_invalidate_both(void);
```

Description

The `cache_invalidate` function and its related functions, `icache_invalidate` and `dcache_invalidate`, invalidate the contents of the processor's instruction and data caches, forcing any data to be re-fetched from memory. Modified data cached in write-back mode is **not** flushed to memory first.

The `cache_invalidate` routine calls its support routines according to the bits set in parameter `cachemask`. The bits have the following meanings.

Bit Set	Meaning
`CPLB_ENABLE_ICACHE`	Invalidate instruction cache
`CPLB_ENABLE_DCACHE`	Invalidate data cache A
`CPLB_ENABLE_DCACHE2`	Invalidate data cache B

A call is made to the appropriate support routine for each bit set. If bits are set to indicate that both data cache A and data cache B must be invalidated, a single call is made to the `dcache_invalidate_both` routine.

On the ADSP-BF535 processor, `cache_invalidate` is called by the default start-up code on reset, and is passed the value of `___cplb_ctrl` as its parameter. Thus, each enabled cache is invalidated during start-up. On other Blackfin processors, the caches automatically reset to the "invalidated" state, and no call is necessary, nor performed.

The `dcache_invalidate` routine only invalidates a single data cache, selected by its `a_or_b` parameter:

a_or_b Value	Meaning
CPLB_INVALIDATE_A	Invalidate data cache A
CPLB_INVALIDATE_B	Invalidate data cache B

The `dcache_invalidate_both` routine invalidates both data cache A and data cache B. On the ADSP-BF535 processor, it is implemented by calling `dcache_invalidate` for each data cache in turn. On other Blackfin processors, the routine toggles the bits of the `DMEM_CONTROL` register to invalidate all contents of both data caches at once, and is considerably faster than calling `dcache_invalidate` for each data cache separately.

Error Conditions

The cache invalidation routines do not return an error condition.

Example

```
#include <cplbtab.h>

void clean_cache(int which)
{
   switch (which) {
   case 1:
         icache_invalidate();
         break;
   case 2:
         dcache_invalidate(CPLB_INVALIDATE_A);
         break;
```

```
    case 4:
          dcache_invalidate(CPLB_INVALIDATE_B);
          break;
    case 6:
          dcache_invalidate_both();
          break;
    default:
          cache_invalidate(__cplb_ctrl);
          break;
    }
}
```

See Also

flush_data_cache

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 4-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

ceil

Ceiling

Synopsis

```
#include <math.h>

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

Description

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

Error Conditions

The ceiling functions do not return an error condition.

Example

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

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

See Also

floor

clearerr

Clear file or stream error indicator

Synopsis

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

Description

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

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

Error Conditions

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

Example

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

See Also

feof, ferror

clock

Processor time

Synopsis

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

Description

The `clock` function returns the number of processor cycles that have elapsed since an arbitrary starting point. The function returns the value `(clock_t) -1`, if the processor time is not available or if it cannot be represented. The result returned by the function may be used to calculate the processor time in seconds by dividing it by the macro `CLOCKS_PER_SEC`. For more information, see "time.h" on page 3-36. An alternative method of measuring the performance of an application is described in "Measuring Cycle Counts" on page 4-64.

Error Conditions

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

Example

```
#include <time.h>

time_t start_time,stop_time;
double time_used;

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

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

See Also

No related function.

cos

Cosine

Synopsis

```
#include <math.h>

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

fract16 cos_fr16 (fract16 x);
fract32 cos_fr32 (fract32 x);

_Fract cos_fx16 (_Fract 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.

Error Conditions

The cosine functions do not return an error condition.

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, 3/2 π]

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.

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 countlsr(fract f);
int countlsk(accum a);

int countlshr(short fract f);
int countlshk(short accum a);

int countlslr(long fract f);
int countlslk(long accum a);

int countlsur(unsigned fract f);
int countlsuk(unsigned accum a);

int countlsuhr(unsigned short fract f);
int countlsuhk(unsigned short accum a);

int countlsulr(unsigned long fract f);
int countlsulk(unsigned long accum a);
```

Description

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

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

Error Conditions

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

Example

```
#include <stdfix.h>
int n;
n = countlsk(-12.5k);            /* n == 4 */
n = countlsulr(0.125ulr);        /* n == 2 */

#if defined(_C99)
n = countlsfx(-12.5k);           /* n == 4 */
n = countlsfx(0.125ulr);         /* n == 2 */
#endif
```

See Also

No related functions.

cplb_hdr

Default exception handler for memory-related events

Synopsis

```
#include <cplbtab.h>
void cplb_hdr (void);
```

Description

The `cplb_hdr` routine is the default exception handler, installed by the default start-up code to service CPLB-related events if CPLBs or caching is indicated by the `___cplb_ctrl` variable.

The routine saves the processor context, before examining the exception details to determine the kind of exception raised. If it is an instruction CPLB miss, a data CPLB miss, or a data CPLB write, the routine invokes `cplb_mgr` to handle the event, otherwise it calls the routine `_unknown_exception_occurred`, which is not expected to return.

If `cplb_mgr` indicates a successful handling, the routine returns from the exception, restoring the context as it does. Otherwise, it invokes an appropriate diagnostic routine.

Error Conditions

The `cplb_hdr` routine calls other routines to deal with each of the error codes returned by the `cplb_mgr` routine. By default, these routines are stubs that loop forever—you can replace them with your own routines if you wish to provide more detailed handling.

Table 3-31 lists the error codes and their responses.

Table 3-31. cplb_hdr Error Codes and Responses

Error Code	Response
CPLB_RELOADED	Indicates success; returns
CPLB_NO_ADDR_MATCH	Calls stub _cplb_miss_without_replacement
CPLB_NO_UNLOCKED	Calls stub _cplb_miss_all_locked
CPLB_PROT_VIOL	Calls stub _cplb_protection_violation
others	Loops at label strange_return_from_cplb_mgr

Example

```
#include <cplbtab.h>
#include <sys/exception.h>

void setup_cplb_handling(void) {
     register_handler(ik_exception, (ex_handler_fn)cplb_hdr);
}
```

See also Blackfin/lib/src/libc/crt/basiccrt.s in the VisualDSP++ installation directory.

See Also

cplb_init, cplb_mgr

cplb_init

Initialize CPLBs and caches at start-up

Synopsis

```
#include <cplbtab.h>
void cplb_init(int bitmask);
```

Description

The `cplb_init` routine is called by the default start-up code during processor initialization. It initializes the memory protection hardware and enables caches where requested, according to configuration data in two tables. It is not expected that `cplb_init()` is called from normal user code, nor is it expected that it is called more than once following each processor reset.

The routine's behavior is controlled by the following data structures:

- The `___cplb_ctrl` variable
- The `dcplbs_table[]` array
- The `icplbs_table[]` array

Initially, the routine tests the `___cplb_ctrl` variable to determine whether any of the caches have been enabled when the `.ldf` file has already mapped code or data into the corresponding cache area. If so, this would lead to corrupted code or data; therefore, the `cplb_init` routine aborts by jumping to infinite loops labelled with diagnostic symbols, for example, `l1_code_cache_enabled_when_l1_used_for_code`.

For the ADSP-BF535 processor, if caches are indicated by the `___cplb_ctrl` variable, then the routine invokes the `cache_invalidate` function to first invalidate the caches, so that they are not enabled while containing random bits.

For each of the data and instruction CPLBs, if requested, the `cplb_init` routine copies from one to sixteen entries from the configuration tables, installing the tables' entries into the corresponding registers. For example, `icplbs_table[0]` is copied into `ICPLB_DATA0` and `ICPLB_ADDR0`, and `dcplbs_table[0]` is copied into `DCPLB_DATA0` and `DCPLB_ADDR0`.

The copying is not verbatim; if caches are not requested by the `___cplb_ctrl` variable, cache bits are masked off the values written to the *x*`CPLB_DATA`*n* registers.

If a table has from one to sixteen entries, all of the table's entries are installed, with any unused *x*`CPLB_DATA`*n* registers being marked as "Invalid". If a table contains more entries, then only the first sixteen are installed. It is assumed that an appropriate exception handler was installed to process any CPLB miss exceptions that occur. The `cplb_hdr` routine is an example of such an exception handler.

After installing the CPLB entries from the tables, the `cplb_init` routine modifies the `IMEM_CONTROL` and `DMEM_CONTROL` registers to enable the CPLBs and caches that were indicated. The `cplb_init` routine also sets the following `DMEM_CONTROL` bits:

- The data cache bank select bit is set, according to whether `CPLB_ENABLE_DCBS` is set.
- The `DAG0/1` port preference bits are set to 1 and 0, respectively, to reduce memory access stalls.

Error Conditions

The `cplb_init` routine does not return an error condition. If it encounters an error during initialization, it jumps to a label indicative of the problem.

Example

See the source for the start-up code, in the VisualDSP++ installation directory, under `...Blackfin/lib/src/libc/crt/basiccrt.s`.

See Also

cplb_hdr

cplb_mgr

CPLB management routine for CPLB exceptions

Synopsis

```
#include <cplbtab.h>
int cplb_mgr(int event, int bitmask);
```

Description

The `cplb_mgr` routine manages the active CPLB tables for instructions and data. It is intended to be invoked from an exception handler upon receipt of a CPLB-related `event`. `cplb_hdr`, installed by the default start-up code, is a typical example of such a handler.

The event parameter indicates the action that the routine should take:

Event Value	Required Action
`CPLB_EVT_ICPLB_MISS`	Replace an active instruction CPLB
`CPLB_EVT_DCPLB_MISS`	Replace an active data CPLB
`CPLB_EVT_DCPLB_WRITE`	Mark an existing data CPLB as dirty

To replace an instruction CPLB, the routine determines the faulting address from the processor's registers, and searches the `icplbs_table[]` looking for an entry whose start address and size addresses a region of memory that includes the faulting address. If none is found, the routine returns `CPLB_NO_ADDR_MATCH` to indicate that the faulting address is not covered by any of the entries in `icplbs_table[]`, and is therefore an invalid address.

The routine selects the first active instruction CPLB that is not locked, and shuffles all following instruction CPLBs up, overwriting it, so that the last instruction CPLB is free to be used by the entry to be installed. In this manner, the replacement algorithm is Least-Recently-Installed. If no unlocked instruction CPLBs can be found, the routine returns CPLB_NO_UNLOCKED.

The new instruction CPLB is installed into ICPLB_ADDR15 and ICPLB_DATA15. If the bitmask parameter indicates that the instruction cache is not enabled (that is, if bitmask does not have bit CPLB_ENABLE_ICACHE set), then cache bits are masked off the entry as it is installed.

Replacing a data CPLB follows a similar process, but must also deal with CPLBs that indicate data is to be cached in write-back mode. In this case, the routine first attempts to select a clean data CPLB to evict. If no unlocked clean data CPLB can be found, then the routine falls back on selecting dirty data CPLBs. As for instruction CPLBs, if no unlocked CPLB can be selected, the routine returns CPLB_NO_UNLOCKED.

If it is necessary to evict a dirty data CPLB, the cplb_mgr routine first flushes any modified cache entries corresponding to the victim data CPLB's memory page, forcing the modified data to be written back to secondary memory.

When the new data CPLB is installed, cache bits are masked off if the bitmask parameter does not have either of the CPLB_ENABLE_DCACHE or CPLB_ENABLE_DCACHE2 bits sets.

The cplb_mgr routine is called to handle data CPLB write events when a page is cached in write-back mode, and the first write occurs to a clean page. In this case, the routine locates the active CPLB using the processor's MMRs and verifies that it is a clean, cached, write-back CPLB, and marks the page as dirty. Future writes to the page do not trigger an exception, but now the page has been marked as dirty, so the routine can flush the modified data from the cache if it becomes necessary to evict the page.

If the page indicated by a data CPLB write is not a clean, cached write-back page, this indicates that a protection violation has occurred, for example, a write to a supervisor-only page while in user mode, and, therefore, the routine returns CPLB_PROT_VIOL.

When a data CPLB is installed during the eviction process, pages that are cached in write-back mode are not forced to be marked as clean. This is because there is a performance trade-off that the application designer can exploit if the expectation is that data CPLB miss exceptions are very rare. A dirty data CPLB is expensive to evict, because of the cost of flushing modified data to secondary memory, but if no eviction is ever expected, this is irrelevant. In contrast, if a page is expected to be modified but never flushed, a clean data CPLB will pay the cost of a data CPLB write exception on first write. Therefore, the designer may choose to pre-mark pages as dirty, with the expectation that the CPLB eviction process will never occur.

Error Conditions

The `cplb_mgr` routine returns the following values:

Value	Meaning
`CPLB_RELOADED`	The event was serviced successfully
`CPLB_NO_ADDR_MATCH`	There is no entry in the appropriate `cplbs_table[]` that covers the faulting address.
`CPLB_NO_UNLOCKED`	All the active CPLBs are marked as “locked” and could not be evicted
`CPLB_PROT_VIOL`	A protection violation occurred

Example

```
#include <cplbtab.h>

void replace_dcplb(void) {
     int r = cplb_mgr(CPLB_EVT_DCPLB_MISS, __cplb_ctrl);
     if (r == CPLB_RELOADED)
          printf("Success\n");
     else
```

```
        printf("Failed to replace Data CPLB\n");
}
```

See also `Blackfin/lib/src/libc/crt/cplbhdr.s` in the VisualDSP++ installation directory.

See Also

cplb_hdr, cplb_init

ctime

Convert calendar time into a string

Synopsis

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

Description

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

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

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

Error Conditions

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

Example

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

time_t cal_time;

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

See Also

asctime, gmtime, localtime, time

difftime

Difference between two calendar times

Synopsis

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

Description

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

Error Conditions

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

Example

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

time_t cal_time1;
time_t cal_time2;
double time_diff;

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

```
else
    time_diff = difftime(cal_time2,cal_time1);
```

See Also

time

disable_data_cache

Disable processor data caches and CPLBs

Synopsis

```
#include <cplbtab.h>
void disable_data_cache(void);
```

Description

The `disable_data_cache` function disables the processor's data caches and data CPLBs.

> The `disable_data_cache` function does not flush back to memory any modified data in the cache that is cached in write-back mode. To flush any such data, use the `flush_data_cache` or `flush_data_buffer` routines.

Error Conditions

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

Example

```
#include <cplbtab.h>

void cache_off(void)
{
      disable_data_cache();
}
```

See Also

cache_invalidate, enable_data_cache, flush_data_cache

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);
int divik(int numer, accum denom);

long int divilr(long int numer, long fract denom);
long int divilk(long int numer, long accum denom);

unsigned int diviur(unsigned int numer, unsigned fract denom);
unsigned int diviuk(unsigned int numer, unsigned accum denom);

unsigned long int diviulr(unsigned long int numer,
                          unsigned long fract denom);
unsigned long int diviulk(unsigned long int numer,
                          unsigned long accum denom);
```

Description

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

Error Conditions

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

Example

```
#include <stdfix.h>
int quo;
unsigned long int ulquo;
quo = divik(125, -12.5k);                  /* quo == -10 */
ulquo = diviulr(125, 0.125ulr);          /* ulquo == 1000 */
```

See Also

fxdivi, idivfx

enable_data_cache

Turn on one or both data caches

Synopsis

```
#include <cplbtab.h>
void enable_data_cache(int bitmask);
```

Description

The `enable_data_cache` function enables one or both of the processor's data caches, as indicated by the `bitmask` parameter:

Bit Set	Meaning
`CPLB_ENABLE_CPLBS` or `CPLB_ENABLE_DCPLBS`	Enable data CPLBs
`CPLB_ENABLE_DCACHE`	Enable data cache A
`CPLB_ENABLE_DCACHE2`	Enable data cache B

The bits are set in the following order:

1. The `CPLB_ENABLE_CPLBS` or `CPLB_ENABLE_DCPLBS` bits must be set. If neither bit is set, the function exits without changing `DMEM_CONTROL`. If one or both of these bits is set, data CPLBs are enabled.
2. If caching is to be enabled, `CPLB_ENABLE_DCACHE` must be set. If so, data cache A is enabled. However, `CPLB_ENABLE_CPLBS` or `CPLB_ENABLE_DCPLBS` must be set first.
3. If both data caches are to be enabled, `CPLB_ENABLE_DCACHE2` must also be set; if so, both data cache A and data cache B are enabled. `CPLB_ENABLE_DCACHE2` is ignored if `CPLB_ENABLE_DCACHE` is not set, as the processor does not support data cache B in isolation.

Valid data CPLBs must already be installed in the DCPLB active table before calling this function; the default start-up code ensures this is the case. If DCPLB misses are a possibility, a suitable exception handler, such as `cplb_hdr`, must be installed by the default start-up code.

The data caches may only be enabled if their memory space has been left free for cache use. If any data has been mapped to this space by the `.ldf` file, the data will be corrupted by the cache's operation, and undefined behavior will result.

Error Conditions

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

Example

```
#include <cplbtab.h>

void cache_on(int howmany)
{
      int bitmask = __cplb_ctrl;
      if (howmany == 1)
          bitmask &= ~CPLB_ENABLE_DCACHE2;
      enable_data_cache(bitmask);
}
```

See Also

cplb_hdr, cplb_init, disable_data_cache

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

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

Example

```
#include <stdlib.h>

exit(EXIT_SUCCESS);
```

See Also

abort, atexit

exp

Exponential

Synopsis

```
#include <math.h>

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

Description

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

Error Conditions

The input argument `x` for `expf` must be in the domain [-87.33, 88.72], and the input argument for `expd` must be in the domain [-708.39, 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`.

Error Conditions

The `fabs` functions do not return error conditions.

Example

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

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

See Also

abs, absfx, labs

fclose

Close a stream

Synopsis

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

Description

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

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

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

Error Conditions

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

Example

```
#include <stdio.h>
void example(char* fname)
{
   FILE *fp;
   fp = fopen(fname, "w+");
   /* Do some operations on the file */
   fclose(fp);
}
```

See Also

fopen

feof

Test for end of file

Synopsis

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

Description

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

Error Conditions

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

Example

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

See Also

clearerr, ferror

ferror

Test for read or write errors

Synopsis

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

Description

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

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

Error Conditions

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

Example

```
#include <stdio.h>
void test_for_error(FILE *fp)
{
   if (ferror(fp))
      printf("Error with read/write to stream\n");
   else
      printf("read/write to stream OKAY\n");
}
```

See Also

clearerr, feof

fflush

Flush a stream

Synopsis

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

Description

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

Upon successful completion the `fflush` function returns zero.

Error Conditions

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

Example

```
#include <stdio.h>
void flush_all_streams(void)
{
   fflush(NULL);
}
```

See Also

fclose

fgetc

Get a character from a stream

Synopsis

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

Description

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

Upon successful completion, the `fgetc` function will return 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 will return zero.

Error Conditions

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

Example

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

```
        printf("fsetpos failed\n");
    }
}
```

See Also

fsetpos, ftell, fseek, rewind

fgets

Get a string from a stream

Synopsis

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

Description

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

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

Error Conditions

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

Example

```
#include <stdio.h>
char buffer[20];
void read_into_buffer(FILE *fp)
{
   char *str;

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

```
    }
}
```

See Also

fgetc, getc, gets

floor

Floor

Synopsis

```
#include <math.h>

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

Description

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

Error Conditions

The `floor` functions do not return error conditions.

Example

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

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

See Also

ceil

flush_data_cache

Flush modified data from cache to memory

Synopsis

```
#include <cplbtab.h>

void flush_data_cache(void);
void flush_data_buffer(void *start, void *end, int invalidate);
```

Description

The `flush_data_cache` function may be used when the processor's data caches are enabled, and some data is being cached in write-back mode. In this mode, modified data is held in the cache, and is not written back to memory immediately, thus saving the cost of an external memory access. When data is cached in this mode, it may be necessary to ensure that any modified data has been flushed to memory, so that external systems can access it. DMA transfers and dual-core accesses are common cases where write-back mode data would need to be flushed.

The `flush_data_cache` function flushes all modified data from the cache to memory. It does so by traversing the table of active data CPLBs, looking for valid entries that indicate a write-back page that has been modified (that is, the dirty flag has been set). For each page encountered, the function flushes the modified data in the page.

The `flush_data_buffer` function may be used when individual areas of memory need to be flushed from the cache. The function flushes data cache addresses from `start` to `end` inclusive. Additional data addresses may also be flushed, since the function flushes entire cache lines.

If the `invalidate` parameter is non-zero, `flush_data_buffer` also invalidates the cache entries, forcing the next data access to re-fetch the data from memory. This is useful if the buffer is being updated by an external activity, such as DMA.

Error Conditions

The `flush_data_cache` and `flush_data_buffer` functions do not return an error condition.

Example

```
#include <cplbtab.h>

void do_flush(void)
{
   if (__cplb_ctrl & (CPLB_ENABLE_DCACHE|CPLB_ENABLE_DCACHE2))
         flush_data_cache();
}

char *buffer;
int buffer_len;
void inv_buffer(void) {
   flush_data_buffer(buffer, buffer+buffer_len, 1);
}
```

See Also

cache_invalidate

fmod

Floating-point modulus

Synopsis

```
#include <math.h>

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

Description

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

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

Error Conditions

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

Example

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

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

See Also

div, ldiv, modf

fopen

Open a file

Synopsis

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

Description

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

Valid selections for `mode` are specified 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.

mode	Selection
`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: 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 `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`.

Length	Action
`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

Specifier	Conversion
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 [-]0x*h.hhhh*p±*d* where there is one hexadecimal digit before the period. The a|A conversion specifiers always contain a minimum of one digit for the exponent.

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

The f|F conversion 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 `fputc` function writes the argument `ch` to the output stream pointed to by `stream` and advances the file position indicator. The argument `ch` is converted to an `unsigned char` before it is written.

If the `fputc` function is successful then it will return the value that was written to the stream.

Error Conditions

If the `fputc` function is not successful, `EOF` is returned.

Example

```
#include <stdio.h>

void fputc_example(FILE* fp)
{
   /* put the character 'i' to the stream pointed to by fp */
   int res = fputc('i', fp);
   if (res != 'i')
      printf("fputc failed\n");
}
```

See Also

putc

fputs

Put a string on a stream

Synopsis

```
#include <stdio.h>
int fputs(const char *string, FILE *stream);
```

Description

The `fputs` function writes the string pointed to by `string` to the output stream pointed to by `stream`. The `NUL` terminating character of the string will not be written to `stream`.

If the call to `fputs` is successful, the function will return a non-negative value.

Error Conditions

The `fputs` function will return `EOF` if a write error occurred.

Example

```
#include <stdio.h>

void fputs_example(FILE* fp)
{
   /* put the string "example" to the stream pointed to by fp */
   char *example = "example";
   int res = fputs(example, fp);
   if (res == EOF)
      printf("fputs failed\n");
}
```

See Also

puts

fread

Buffered input

Synopsis

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

Description

The `fread` function reads into an array pointed to by `ptr` up to a maximum of `n` items of data from `stream`, where 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` will return zero and does not affect the state of `stream`.

When the stream has been opened as a binary stream, the Analog Devices I/O library may choose to bypass the I/O buffer and transmit data from an external device directly into the program, particularly when the buffer size (as defined by the macro `BUFSIZ` in the `stdio.h` header file or controlled by the function `setvbuf`) is smaller than the number of characters to be transferred. 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` will return 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

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

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 file descriptor

Synopsis

```
#include <stdio.h>
FILE *freopen(const char *fname, const char *mode, FILE *stream);
```

Description

The `freopen` function opens the file specified by `fname` and associates it with the stream pointed to by `stream`. The mode argument has the same effect as described in `fopen` (see "fopen" on page 3-152 for more information on the `mode` argument).

Before opening the new file, the `freopen` function will first attempt to flush the stream and close any file descriptor associated with `stream`. Failure to flush or close the file successfully is ignored. Both the error and `EOF` indicators for `stream` are cleared.

The original stream will always be closed regardless of whether the opening of the new file is successful or not.

Upon successful completion, the `freopen` function returns the value of `stream`.

Error Conditions

If `freopen` is unsuccessful, a `NULL` pointer is returned.

Example

```
#include <stdio.h>

void freopen_example(FILE* fp)
{
```

```
    FILE *result;
    char *newname = "newname";

    /* reopen existing file pointer for reading file "newname" */
    result = freopen(newname, "r", fp);
    if (result == fp)
       printf("%s reopened for reading\n", newname);
    else
       printf("freopen not successful\n");
}
```

See Also

fclose, fopen

frexp

Separate fraction and exponent

Synopsis

```
#include <math.h>

float frexpf (float 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 ±[½, 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.

Error Conditions

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

Example

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

y = frexp  (2.0, &exponent);     /* y = 0.5, exponent = 2  */
x = frexpf (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 finds a non-whitespace character. If the format specification contains a sequence of ordinary characters, then `fscanf` will continue to read the next characters in the input stream until the input data does not match the sequence of characters in the format. At this point `fscanf` will fail, and the differing and subsequent characters in the input stream will not be read.

The `%` character in the format string introduces a conversion specification. A conversion specification has the following form:

```
% [*] [width] [length] type
```

A conversion specification always starts with the `%` character. It may optionally be followed by an asterisk (*) character, which indicates that the result of the conversion is not to be saved. In this context, the asterisk character is known as the *assignment-suppressing character*. The optional token `width` represents a non-zero decimal number and specifies the maximum field width. 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` (see “-full-io” on page 1-40) or `-fixed-point-io` switches (see “-fixed-point-io” on page 1-38).

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 will contain "a" */
    fscanf (fp, "%[aeiou]", string);

    /* For input "drying", string will contain "dry" */
    fscanf (fp, "%[^aeiou]", string);
}
```

See Also

scanf, sscanf

fseek

Reposition a file position indicator in a stream

Synopsis

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
```

Description

The `fseek` function sets the file position indicator for the stream pointed to by `stream`. The position within the file is calculated by adding the offset to a position dependent on the value of `whence`. The valid values and effects for `whence` are as follows:

whence	Effect
SEEK_SET	Set the position indicator to be equal to `offset` bytes from the beginning of `stream`.
SEEK_CUR	Set the new position indicator to current position indicator for `stream` plus `offset`.
SEEK_END	Set the position indicator to `EOF` plus `offset`.

Using `fseek` to position a text stream is only valid if either `offset` is zero, or if `whence` is `SEEK_SET` and `offset` is a value that was previously returned by `ftell`.

ⓘ 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.

ⓘ 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` function clears the `EOF` indicator for `stream` and undoes any effects of `ungetc` on the same stream.

The `fsetpos` function returns zero if it is successful.

Error Conditions

If the `fsetpos` function is unsuccessful, the function returns a non-zero value.

Example

Refer to "fgetpos" on page 3-144 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.

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, the `ftell` function returns the current value of the file position indicator on the stream.

Error Conditions

If the `ftell` function is unsuccessful, a value of -1 is returned.

Example

See `fseek` for an example.

See Also

fseek

fwrite

Buffered output

Synopsis

```
#include <stdio.h>

size_t fwrite(const void *ptr, size_t size, size_t n,
              FILE *stream);
```

Description

The `fwrite` function writes to the output `stream` up to `n` items of data from the array pointed by `ptr`. An item of data is defined as a sequence of characters of size `size`. The write will complete once `n` items of data have been written to the stream. The file position indicator for `stream` is advanced by the number of characters successfully written.

When the stream has been opened as a binary stream, the Analog Devices I/O library may choose to bypass the I/O buffer and transmit data from the program directly to the external device, particularly when the buffer size (as defined by the macro `BUFSIZ` in the `stdio.h` header file, or controlled by the function `setvbuf`) is smaller than the number of characters to be transferred. If 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 will return the number of items written.

Error Conditions

If the `fwrite` function is unsuccessful, it will return the number of elements successfully written which will be less than `n`.

Example

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

fract rbits(int_r_t b);
accum kbits(int_k_t b);

short fract hrbits(int_hr_t b);
short accum hkbits(int_hk_t b);

long fract lrbits(int_lr_t b);
long accum lkbits(int_lk_t b);

unsigned short fract uhrbits(uint_uhr_t b);
unsigned short accum uhkbits(uint_uhk_t b);

unsigned fract urbits(uint_ur_t b);
unsigned accum ukbits(uint_uk_t b);

unsigned long fract ulrbits(uint_ulr_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

The `fxbits` family of functions do not return an error condition. If the input integer value does not fit in the number of bits of the fixed-point result type, the result is saturated to the largest or smallest fixed-point value.

Example

```
#include <stdfix.h>
accum k;
unsigned long fract ulr;
k = kbits(-0x640000000ll);              /* 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);
accum kdivi(int numer, int denom);

long fract lrdivi(long int numer, long int denom);
long accum lkdivi(long int numer, long int denom);

unsigned fract urdivi(unsigned int numer, unsigned int denom);
unsigned accum ukdivi(unsigned int numer, unsigned int denom);

unsigned long fract ulrdivi(unsigned long int numer,
                            unsigned long 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.

Error Conditions

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

Example

```
#include <stdio.h>

char use_getc(FILE *fp)
{
   char ch;
   if ((ch = getc(fp)) == EOF) {
      printf("Read End-of-file\n");
      return (char)-1;
   } else {
      return ch;
   }
}
```

See Also

fgetc

getchar

Get a character from `stdin`

Synopsis

```
#include <stdio.h>
int getchar(void);
```

Description

The `getchar` function is functionally the same as calling the `getc` function with `stdin` as its argument. A call to `getchar` will return the next single character from the standard input stream. The `getchar` function also advances the standard input's current position indicator.

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.

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" on page 3-36.

The broken-down time is returned by `gmtime` as a pointer to static memory, which may be overwritten by a subsequent call to either `gmtime`, or to `localtime`.

Error Conditions

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

Example

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

time_t cal_time;
struct tm *tm_ptr;

cal_time = time(NULL);
if (cal_time != (time_t) -1) {
    tm_ptr = gmtime(&cal_time);
    printf("The year is %4d\n",1900 + (tm_ptr->tm_year));
}
```

See Also

localtime, mktime, time

heap_calloc

Allocate and initialize memory 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 will contain `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 Multiple Heaps" on page 1-423 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 was allocated and returns the memory to this heap. For more information on creating multiple run-time heaps, refer to "Using Multiple Heaps" on page 1-423.

Error Conditions

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

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 index);
```

Description

The `heap_init` function re-initializes a heap, emptying the free list, and discarding all records within the heap. Because the function discards any records 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.

> 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 run-time

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 will return the heap index for the heap once it has been successfully initialized.

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" on page 1-423 for more information.

To re-initialize a heap that is already installed, use the `heap_init` function (on page 3-196).

Error Conditions

The `heap_install` function returns -1 if the heap was not initialized successfully. This may occur, for example, if the `__heaps` table could not be sufficiently resized, if a heap with the specified `userid` already exists, or if the new heap is too small.

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" on page 1-423.

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" on page 1-423.

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 will return 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 idx);
```

Description

The `heap_space_unused` function returns the total free space in bytes for the heap with index `idx`.

Note that calling `heap_malloc(idx,heap_space_unused(idx))` 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 `idx` 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);
int idivk(accum numer, accum denom);

long int idivlr(long fract numer, long fract denom);
long int idivlk(long accum numer, long accum denom);

unsigned int idivur(unsigned fract numer, unsigned fract denom);
unsigned int idivuk(unsigned accum numer, unsigned accum denom);

unsigned long int idivulr(unsigned long fract numer,
                          unsigned long fract denom);
unsigned long int idivulk(unsigned long accum numer,
                          unsigned long accum denom);
```

Description

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

Error Conditions

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

Example

```
#include <stdfix.h>
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

interrupt

Define interrupt handling

Synopsis

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

Description

The `interrupt` function determines how a signal received during program execution is handled. The `interrupt` function executes the function pointed to by `func` at every signal `sig`. The `signal` function executes the function only once.

The `func` argument must be one of the values listed in Table 3-32. The interrupt function causes the receipt of the signal number `sig` to be handled in one of the ways shown in Table 3-32.

Table 3-32. Interrupt Handling: `func` Argument

Func Value	Action
`SIG_DFL`	The signal is enabled, but ignored when it occurs.
`SIG_IGN`	The signal is disabled.
`Function address`	The signal is enabled, and the function is called when the signal occurs.

The function pointed to by `func` is executed each time the `interrupt` is received. The `interrupt` function must be called with the `SIG_IGN` argument to disable interrupt handling. The `sig` argument may be any of the signals shown in Table 3-33 on page 3-262 which lists the supported signals in interrupt priority order from highest to lowest.

When the function pointed to by `func` is executed, the parameter `val` is set to the number of the signal that has been received. So if `func` is a signal handler used for various signals, `func` can find out which signal it is handling.

The function pointed to by `func` must not be defined using `#pragma interrupt`; the `#pragma interrupt` functions are registered using `register_handler_ex()` or `register_handler()` instead.

Refer to "Interrupt Handler Support" on page 1-365 for more information.

See Also

raise, register_handler, register_handler_ex, signal

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.

Error Conditions

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

Example

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

for (ch=0; ch<=0x7f; ch++) {
   printf("%#04x", ch);
   printf("%3s", isalnum(ch) ? "alphanumeric" : "");
   putchar('\n');
}
```

See Also

isalpha, isdigit

isalpha

Detect alphabetic character

Synopsis

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

Description

The `isalpha` function determines 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.

Error Conditions

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

Example

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

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

See Also

isalnum, isdigit

iscntrl

Detect control character

Synopsis

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

Description

The `iscntrl` function determines 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.

Error Conditions

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

Example

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

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

See Also

isalnum, isgraph

isdigit

Detect decimal digit

Synopsis

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

Description

The `isdigit` function determines 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.

Error Conditions

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

Example

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

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

See Also

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.

Error Conditions

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

Example

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

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

See Also

isalnum, iscntrl, isprint

isinf

Test for infinity

Synopsis

```
#include <math.h>

int 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 will return a non-zero value.

Error Conditions

The `isinf` functions do not return or set any error conditions.

Example

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

Error Conditions

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

Example

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

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

See Also

isalpha, isupper

isnan

Test for Not-a-Number (NAN)

Synopsis

```
#include <math.h>

int isnanf(float x);
int isnan(double x);
int isnand (long double x);
```

Description

The `isnan` functions return a zero if the argument is not set to an IEEE `NaN`; otherwise, the functions return a non-zero value.

Error Conditions

The `isnan` functions do not return or set any error conditions.

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.

Error Conditions

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

Example

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

for (ch=0; ch<=0x7f; ch++) {
   printf("%#04x", ch);
   printf("%3s", isprint(ch) ? "printable" : "");
   putchar('\n');
}
```

See Also

isgraph, isspace

ispunct

Detect punctuation character

Synopsis

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

Description

The `ispunct` function determines 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.

Error Conditions

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

Example

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

for (ch=0; ch<=0x7f; ch++) {
   printf("%#04x", ch);
   printf("%3s", ispunct(ch) ? "punctuation" : "");
   putchar('\n');
}
```

See Also

isalnum

isspace

Detect whitespace character

Synopsis

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

Description

The `isspace` function determines 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.

Error Conditions

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

Example

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

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

See Also

iscntrl, isgraph

isupper

Detect uppercase character

Synopsis

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

Description

The `isupper` function determines 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.

Error Conditions

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

Example

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

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

See Also

isalpha, islower

isxdigit

Detect hexadecimal digit

Synopsis

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

Description

The `isxdigit` function determines 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.

Error Conditions

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

Example

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

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

See Also

isalnum, isdigit

_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` characters 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`. Both functions share the following restrictions:

- `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 dual-core processors, `instrp` must correspond to the core calling the function.

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`.

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 *)0xFFA00000;
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 *)0xFFA00000;
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.

Error Conditions

The `labs` and `llabs` functions do not return an error condition.

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`.

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`

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" on page 3-36. This implementation of the header file does not support the Daylight Saving flag nor does it support time zones and, thus, `localtime` is equivalent to the `gmtime` function.

The broken-down time is returned by `localtime` as a pointer to static memory, which may be overwritten by a subsequent call to either `localtime` or to `gmtime`.

Error Conditions

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

Example

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

time_t cal_time;
struct tm *tm_ptr;

cal_time = time(NULL);
if (cal_time != (time_t) -1) {
    tm_ptr = localtime(&cal_time);
```

```
    printf("The year is %4d\n",1900 + (tm_ptr->tm_year));
}
```

See Also

asctime, gmtime, mktime, time

log

Natural logarithm

Synopsis

```
#include <math.h>

float logf (float x);
double log (double x);
long double logd (long double x);
```

Description

The natural logarithm functions compute the natural (base e) logarithm of their argument.

Error Conditions

The natural logarithm functions return `-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.

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.

> 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

The longjmp function does not return an error condition.

Example

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

jmp_buf env;
int res;

void 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.

Error Conditions

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

Example

```
#include <string.h>
char *ptr;

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

See Also

strchr, strrchr

memcmp

Compare objects

Synopsis

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

Description

The `memcmp` function compares the first `n` characters of the objects pointed to by `s1` and `s2`. 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.

Error Conditions

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

Example

```
#include <string.h>
char *string1 = "ABC";
char *string2 = "BCD";
int result;

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

See Also

strcmp, strcoll, strncmp

memcpy

Copy characters from one object to another

Synopsis

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

Description

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

> 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`.

Error Conditions

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

Example

```
#include <string.h>
char *a = "SRC";
char *b = "DEST";
memcpy (b, a, 3);       /*  b="SRCT"  */
```

See Also

memmove, strcpy, strncpy

memmove

Copy characters between overlapping objects

Synopsis

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

Description

The `memmove` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The entire object is copied correctly even if the objects overlap.

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

Error Conditions

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

Example

```
#include <string.h>
char *ptr, *str = "ABCDE";

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

See Also

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`.

Error Conditions

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

Example

```
#include <string.h>
char string1[50];
memset(string1, '\0', 50);      /* set string1 to 0 */
```

See Also

memcpy

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 Greenwich Mean Time (UTC).

A broken-down time is a structured variable which is defined in the `time.h` header file as:

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

The various components of the broken-down time are not restricted to the ranges indicated above. The `mktime` function calculates the calendar time from the specified values of the components (ignoring the initial values of `tm_wday` and `tm_yday`) and then “normalizes” the broken-down time forcing each component into its defined range.

If the component tm_isdst is zero, then the mktime function assumes that daylight saving is not in effect for the specified time. If the component is set to a positive value, then the function assumes that daylight saving is in effect for the specified time and will make the appropriate adjustment to the broken-down time. If the component is negative, the mktime function should attempt to determine whether daylight saving is in effect for the specified time but because neither time zones nor daylight saving are supported, the effect will be as if tm_isdst were set to zero.

Error Conditions

The mktime function returns the value (time_t) -1 if the calendar time cannot be represented.

Example

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

static const char *wday[] = {"Sun","Mon","Tue","Wed",
                             "Thu","Fri","Sat","???"};

struct tm tm_time = {0,0,0,0,0,0,0,0,0};

tm_time.tm_year = 2000 - 1900;
tm_time.tm_mday = 1;

if (mktime(&tm_time) == -1)
    tm_time.tm_wday = 7;
printf("%4d started on a %s\n",
    1900 + tm_time.tm_year,
    wday[tm_time.tm_wday]);
```

See Also

gmtime, localtime, time

modf

Separate integral and fractional parts

Synopsis

```
#include <math.h>

float modff (float x, float *intptr);
double modf (double x, double *intptr);
long double modfd (long double x, long double *intptr);
```

Description

The `modf` functions separate the first argument into integral and fractional portions. The fractional portion is returned and the integral portion is stored in the object pointed to by `intptr`. The integral and fractional portions have the same sign as the input.

Error Conditions

The `modf` functions do not return error conditions.

Example

```
#include <math.h>
double y, n;
float m, p;

y = modf (-12.345, &n);    /* y = -0.345, n = -12.0 */
m = modff (11.75, &p);     /* m = 0.75, p = 11.0    */
```

See Also

frexp

mulifx

Multiplication of integer by fixed-point to give integer result

Synopsis

```
#include <stdfix.h>

int mulir(int i, fract f);
int mulik(int i, accum a);

long int mulilr(long int i, long fract f);
long int mulilk(long int i, long accum a);

unsigned int muliur(unsigned int i, unsigned fract f);
unsigned int muliuk(unsigned int i, unsigned accum a);

unsigned long int muliulr(unsigned long int i,
                          unsigned long fract f);
unsigned long int muliulk(unsigned long int i,
                          unsigned long accum a);
```

Description

Given an integer and a fixed-point value, the `muli`*`fx`* family of functions computes the product and returns the closest integer value to the result.

Error Conditions

The `muli`*`fx`* family of functions do not return error conditions.

Example

```
#include <stdfix.h>
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 the string `s` is not a null pointer and if the first character addressed by `s` is not a null character, the function will output the string `s` followed by the character sequence `": "`. The function will then print the message that is associated with the current value of `errno`. Note that the message "`no error`" is used if the value of `errno` is zero.

Error Conditions

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

Example

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

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 `pow` functions compute the value of the first argument raised to the power of the second argument.

Error Conditions

The `pow` 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 will 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" on page 3-154) for a description of the valid format specifiers.

The `printf` function returns the number of characters transmitted.

Error Conditions

If the `printf` function is unsuccessful, a negative value is returned.

Example

```
#include <stdio.h>

void printf_example(void)
{
   int arg = 255;
   /* Output will be "hex:ff, octal:377, integer:255" */
   printf("hex:%x, octal:%o, integer:%d\n", arg, arg, arg);
}
```

See Also

fprintf

putc

Put a character on a stream

Synopsis

```
#include <stdio.h>
int putc(int ch, FILE *stream);
```

Description

The `putc` function writes its argument to the output stream pointed to by `stream`, after converting `ch` from an `int` to an `unsigned char`.

If the `putc` function call is successful, `putc` returns its argument `ch`.

Error Conditions

The stream's error indicator will be set if the call is unsuccessful, and the function will return `EOF`.

Example

```
#include <stdio.h>

void putc_example(void)
{
   /* write the character 'a' to stdout */
   if (putc('a', stdout) == EOF)
      fprintf(stderr, "putc failed\n");
}
```

See Also

fputc

putchar

Write a character to `stdout`

Synopsis

```
#include <stdio.h>
int putchar(int ch);
```

Description

The `putchar` function writes its argument to the standard output stream, after converting `ch` from an `int` to an `unsigned char`. A call to `putchar` is equivalent to calling `putc(ch, stdout)`.

The function is implemented as an inline function if the language dialect is C++; for other C language dialects, it is implemented as a macro if the 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 will return `EOF`.

Example

```
#include <stdio.h>

void putchar_example(void)
{
   /* write the character 'a' to stdout */
   if (putchar('a') == EOF)
```

```
        fprintf(stderr, "putchar failed\n");
    }
```

See Also

putc

puts

Put a string to `stdout`

Synopsis

```
#include <stdio.h>
int puts(const char *s);
```

Description

The `puts` function writes the string pointed to by `s`, followed by a `NEWLINE` character, to the standard output stream `stdout`. The terminating null character of the string is not written to the stream.

If the function call is successful, then the return value is zero or greater.

Error Conditions

The macro `EOF` is returned if `puts` was unsuccessful, and the error indicator for `stdout` will be set.

Example

```
#include <stdio.h>

void puts_example(void)
{
   /* write the string "example" to stdout */
   if (puts("example") < 0)
      fprintf(stderr, "puts failed\n");
}
```

See Also

fputs

qsort

Quicksort

Synopsis

```
#include <stdlib.h>

void qsort (void *base, size_t nelem, size_t size,
            int (*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

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

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 sends the signal `sig` to the executing program. The `raise` function forces interrupts wherever possible and simulates an interrupt otherwise. The `sig` argument must be one of the signals listed in priority order in Table 3-33.

Table 3-33. Raise Function Signals – Values and Meanings

Sig Value	Definition
`SIGEMU`	Emulation trap
`SIGRSET`	Machine reset
`SIGNMI`	Non-maskable interrupt
`SIGEVNT`	Event vectoring
`SIGHW`	Hardware error
`SIGTIMR`	Timer events Note that `SIGALRM` is mapped onto the signal `SIGTIMR`
`SIGIVG7 - SIGIVG15`	Miscellaneous interrupts Note that:`SIGUSR1` is mapped onto the signal `SIGIVG15` `SIGUSR2` is mapped onto the signal `SIGIVG14`
`SIGINT`	Software interrupt
`SIGILL`	Software interrupt
`SIGBUS`	Software interrupt
`SIGFPE`	Software interrupt
`SIGSEGV`	Software interrupt

Table 3-33. Raise Function Signals – Values and Meanings (Cont'd)

Sig Value	Definition
SIGTERM	Software interrupt
SIGABRT	Software interrupt

When an interrupt is forced, the current ISR registered in the event vector table is invoked. Normally, this is a dispatcher installed by `signal()`, which saves the context before invoking the signal handler, and restores it afterwards.

When an interrupt is simulated, `raise()` calls the registered signal handler directly.

Error Conditions

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

Example

```
#include <signal.h>
raise(SIGABRT);
```

See Also

interrupt, signal

rand

Random number generator

Synopsis

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

Description

The `rand` function returns a pseudo-random integer value in the range $[0, 2^{30} - 1]$.

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

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

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 4-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

register_handler

Register event handlers

Synopsis

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

ex_handler_fn register_handler(interrupt_kind kind,
                               ex_handler_fn fn);
```

Description

The `register_handler` function determines how the hardware event `kind` is handled. This is done by registering the function pointed to by `fn` as a handler for the event and updating the `IMASK` register so that interrupt can take effect. The `kind` event is an enumeration identifying each of the hardware events—interrupts and exceptions—accepted by the Blackfin processor.

The `register_handler_ex` function provides an extended and more functional interface than `register_handler`. For more information, see "register_handler_ex" on page 3-270.

For the values for `kind`, refer to "Registering an ISR" on page 1-368. The `fn` must be one of the values listed here.

fn Value	Action
`EX_INT_IGNORE`	The event is disabled; the vector table is unchanged.
`EX_INT_DEFAULT`	The event is disabled; the vector table is cleared.
`Function address`	The event is enabled; the address is entered into the vector table.

The vector table is used by the Blackfin processor to identify instructions to execute when an event occurs. When a given event is raised, and if the event is enabled, the processor begins executing instructions from the address given by the event's entry in the vector table.

No dispatcher is used to invoke `fn`. Therefore, `fn` must be a full event handler. That is, it must save the processor context on entry, restore the context on exit, and return using the machine instruction appropriate to the event type. Therefore, if `fn` is written in C, it must be defined with an appropriate `#pragma` to ensure the compiler generates suitable code. A normal C function is ***not*** suitable for use with `register_handler`. The header file `<sys/exception.h>` provides macros to be used with `register_handler` for prototyping and declaring functions.

The `register_handler` function is a more direct mechanism than `signal` and `interrupt`. The `signal` and `interrupt` functions accept (and require) "normal" C functions, and therefore need to use a dispatcher to invoke the registered function. In contrast, `register_handler` does not use a dispatcher, and so, "normal" C functions are ***not*** suitable for registering with the `register_handler` function.

Note that `register_handler` does not modify the interrupt latch register. Therefore, if `register_handler` is called to install a handler for a latched interrupt, the interrupt handler is called during the execution of `register_handler`. The appropriate bit in the interrupt latch register must be unset by the user if this is undesirable behavior. See the appropriate *Hardware Reference* manual for details of how to do this.

Refer to "Interrupt Handler Support" on page 1-365 for more information.

The function returns a pointer that is in the event vector table for the hardware event `kind` upon entry to `register_handler`.

Example

```
#include <sys/exception.h>
int timer_count = 0;

EX_INTERRUPT_HANDLER(inccount)
{
    timer_count++;
}

main(void)
{
    register_handler(ik_timer, inccount);
}
```

See Also

interrupt, raise, register_handler_ex, signal

register_handler_ex

Register event handlers (extended interface)

Synopsis

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

ex_handler_fn register_handler_ex(interrupt_kind kind,
                                  ex_handler_fn  fn,
                                  int enable);
```

Description

The `register_handler_ex` function determines how the hardware event `kind` is handled. This is done by registering the function pointed to by `fn` as a handler for the event. The `kind` event is an enumeration identifying each of the hardware events interrupts and exceptions accepted by the Blackfin processor.

For the values for `kind`, refer to "Registering an ISR" on page 1-368. The `fn` must be one of the values listed here.

fn Value	Action
`EX_INT_IGNORE`	The event is disabled; the vector table is unchanged.
`EX_INT_DEFAULT`	The event is disabled; the vector table is cleared.
`Function address`	The event is enabled; the address is entered into the vector table.

The vector table is used by the Blackfin processor to identify instructions to execute when an event occurs. When a given event is raised, and if the event is enabled, the processor begins executing instructions from the address given by the events entry in the vector table.

No dispatcher is used to invoke `fn`. Therefore, `fn` must be a full event handler. That is, it must save the processor context on entry, restore the context on exit, and return using the machine instruction appropriate to the event type. Therefore, if `fn` is written in C, it must be defined with an appropriate `#pragma` to ensure that the compiler generates suitable code. A normal C function is not suitable for use with `register_handler_ex`. The header file `<sys/exception.h>` provides macros to be used with `register_handler_ex` for prototyping and declaring functions.

If `fn` is one of the special values shown in the table above, the value of `enable` is ignored, unless `enable == EX_INT_ALWAYS_ENABLE`. The parameter `enable` must be one of the values listed here:

Enable	Action
`EX_INT_DISABLE`	Register `fn`. The interrupt will be disabled.
`EX_INT_ENABLE`	Register `fn`. The interrupt will be enabled.
`EX_INT_KEEP_IMASK`	Register `fn`. The interrupt will remain in the state it was before calling `register_handler_ex` (that is, if it was enabled, it stays enabled).
`EX_INT_ALWAYS_ENABLE`	Install `fn` if `fn != EX_INT_IGNORE` and `fn != EX_INT_DISABLE`. Then enable the interrupt in `IMASK` no matter what the value of `fn` is, and return. Calling `register_handler_ex` with `fn == EX_INT_IGNORE` and `enable == EX_INT_ALWAYS_ENABLE` will enable the hardware event kind without changing the registered handler function.

The `register_handler_ex` function is a more direct mechanism than the `signal` and `interrupt` functions. These functions accept (and require) normal C functions, and therefore need to use a dispatcher to invoke the registered function. In contrast, `register_handler_ex` does not use a dispatcher, and so, normal C functions are not suitable for registering with the `register_handler_ex` function.

The `register_handler_ex` function does not modify the interrupt latch register. Therefore, if `register_handler_ex` is called to install a handler for a latched interrupt, the interrupt handler is called during the execution of `register_handler_ex`. The appropriate bit in the interrupt latch register must be unset by the user if this is undesirable behavior. See the appropriate *Hardware Reference* manual for details of how to do this.

The return value for `register_handler_ex` is the value that was in the event vector table entry for an interrupt of type `kind` when `register_handler_ex` was called.

Refer to "Interrupt Handler Support" on page 1-365 for more information.

The function returns a pointer that is in the event vector table for the hardware event `kind` upon entry to `register_handler`.

Example

```
#include <sys/exception.h>
int timer_count = 0;

EX_INTERRUPT_HANDLER(inccount)
{
    timer_count++;
}

main(void)
{
    /* Register a handler for the ik_timer event and enable it */
    register_handler_ex(ik_timer, inccount, EX_INT_ENABLE);
    /* Disable the ik_timer interrupt   */
    /* keeping the handler in the table */
    register_handler_ex(ik_timer, EX_INT_IGNORE,
                                  EX_INT_DISABLE);
```

```
    /* Re-enable the ik_timer_interrupt,       */
    /* using the existing handler in the table */
    register_handler_ex(ik_timer, EX_INT_IGNORE,
                                  EX_INT_ALWAYS_ENABLE);
}
```

See Also

interrupt, raise, register_handler_ex, signal

remove

Remove file

Synopsis

```
#include <stdio.h>
int remove(const char *filename);
```

Description

The `remove` function removes the file whose name is `filename`. After the function call, `filename` will no longer be accessible.

The `remove` function is only supported under the default device driver supplied by the VisualDSP++ simulator and EZ-KIT Lite evaluation system and it only operates on the host file system.

The `remove` function returns zero on successful completion.

Error Conditions

If the `remove` function is unsuccessful, a non-zero value is returned.

Example

```
#include <stdio.h>

void remove_example(char *filename)
{
   if (remove(filename))
      printf("Remove of %s failed\n", filename);
   else
      printf("File %s removed\n", filename);
}
```

See Also

rename

rename

Rename a file

Synopsis

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

Description

The `rename` function 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 only supported under the default device driver supplied by the VisualDSP++ simulator and EZ-KIT Lite evaluation system and it only operates on the host file system.

If `rename` is successful, a value of zero is returned.

Error Conditions

If `rename` fails, the file named `oldname` is unaffected and a non-zero value is returned.

Example

```
#include <stdio.h>

void rename_file(char *new, char *old)
{
   if (rename(old, new))
      printf("rename failed for %s\n", old);
   else
      printf("%s now named %s\n", old, new);
}
```

See Also

remove

rewind

Reset file position indicator in a stream

Synopsis

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

Description

The `rewind` function sets the file position indicator for `stream` to the beginning of the file. This is equivalent to using the `fseek` routine in the following manner:

```
fseek(stream, 0, SEEK_SET);
```

with the exception that `rewind` will also clear the error indicator.

Error Conditions

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

Example

```
#include <stdio.h>
char buffer[20];
void rewind_example(FILE *fp)
{
   /* write "a string" to a file */
   fputs("a string", fp);
   /* rewind the file to the beginning */
   rewind(fp);
   /* read back from the file - buffer will be "a string" */
   fgets(buffer, sizeof(buffer), fp);
}
```

See Also

fseek

roundfx

Round a fixed-point value to a specified precision

Synopsis

```
#include <stdfix.h>

fract roundr(fract f, int n);
accum roundk(accum a, int n);

short fract roundhr(short fract f, int n);
short accum roundhk(short accum a, int n);

long fract roundlr(long fract f, int n);
long accum roundlk(long accum a, int n);

unsigned fract roundur(unsigned fract f, int n);
unsigned accum rounduk(unsigned accum a, int n);

unsigned short fract rounduhr(unsigned short fract f, int n);
unsigned short accum rounduhk(unsigned short accum a, int n);

unsigned long fract roundulr(unsigned long fract f, int n);
unsigned long accum roundulk(unsigned long accum a, int n);
```

Description

The `round`*`fx`* family of functions round a fixed-point value to the number of fractional bits specified by the second argument. The rounding is round-to-nearest. If the rounded result is out of range of the result type, the result saturated to the maximum or minimum fixed-point value.

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

Error Conditions

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

Example

```
#include <stdfix.h>
accum a;
long fract f;
a = roundhk(-12.51k, 1);                /* a == 12.5k         */
f = roundulr(0x12345678p-32ulr, 16);    /* f == 0x12340000ulr */

#if defined(_C99)
a = roundfx(-12.51k, 1);                /* a == 12.5k         */
f = roundfx(0x12345678p-32ulr, 16);     /* f == 0x12340000ulr */
#endif
```

See Also

No related functions.

scanf

Convert formatted input from `stdin`

Synopsis

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

Description

The `scanf` function reads from the standard input stream `stdin`, interprets the inputs according to `format`, and stores the results of the conversions in 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" on page 3-169.

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

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

Example

```
#include <stdio.h>
#include <stdlib.h>
void* allocate_buffer_from_heap(FILE* fp)
{
   /* Allocate a buffer from the heap for the file pointer */
   void* buf = malloc(BUFSIZ);
   if (buf != NULL)
      setbuf(fp, buf);
   return buf;
}
```

See Also

setvbuf

setjmp

Define a run-time label

Synopsis

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Description

The `setjmp` function saves the calling environment in the `jmp_buf` argument. The effect of the call is to declare a run-time label that can be jumped to via a subsequent call to `longjmp`.

When `setjmp` is called, it immediately returns with a result of zero to indicate that the environment has been saved in the `jmp_buf` argument. If, at some later point, `longjmp` is called with the same `jmp_buf` argument, `longjmp` restores the environment from the argument. The execution 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.

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

The label `setjmp` does not return an error condition.

Example

See the code example for “longjmp” on page 3-236.

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 will return 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 val))) (int);
```

Description

The `signal` function determines how a signal received during program execution is handled. This function causes a response to any single occurrence of an interrupt. The `sig` argument must be one of the signals listed in priority order in Table 3-33 on page 3-262.

Event handlers may also be installed directly; for more information, refer to “Interrupt Handler Support” on page 1-365. The default run-time header installs event handlers that invoke handlers registered by `signal()`.

The `signal` function installs a dispatcher ISR into the event vector table, and enables the relevant event. When the event occurs, the dispatcher saves the processor context before the invoked `func`, and restores the context afterwards.

- If the function is `SIG_DFL`, the signal is enabled, but ignored when it occurs.
- If the function is `SIG_IGN`, the signal is disabled.

When the function pointed to by `func` is executed, the parameter `val` is set to the number of the signal that has been received. Thus, it has the same value as `sig`, assuming that for each signal `sig`, a unique function is registered.

The function pointed to by `func` must not be defined using `#pragma interrupt`; the `#pragma interrupt` functions are registered using `register_handler_ex()` or `register_handler()` instead. Refer to "Registering an ISR" on page 1-368 and "ISRs and ANSI C Signal Handlers" on page 1-370 for more information.

See Also

interrupt, raise, register_handler_ex, register_handler

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 `sin` functions return the sine of the argument. Both the argument `x` and the results returned by the functions are in radians.

The `sin_fr16`, `sin_fr32`, `sin_fx16` and `sin_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 `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.

Error Conditions

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

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 `sinh` functions return the hyperbolic sine of `x`.

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" on page 3-154 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 will contain all of the formatted text if the return value is not negative and is also less than `n`.

Error Conditions

The `snprintf` function returns a negative value if a formatting error occurred.

Example

```
#include <stdio.h>
#include <stdlib.h>
extern char *make_filename(char *name, int id)
{
   char *filename_template = "%s%d.dat";
   char *filename = NULL;

   int len = 0;
   int r;                   /* return value from snprintf   */

   do {
      r = snprintf(filename,len,filename_template,name,id);
      if (r < 0)            /* formatting error?            */
         abort();
      if (r < len)          /* was complete string written? */
         return filename;   /* return with success          */
      filename = realloc(filename,(len=r+1));
   } while (filename != NULL);
   abort();
}
```

See Also

fprintf, sprintf, vsnprintf

space_unused

Space unused in heap

Synopsis

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

Description

The `space_unused` function returns the 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 will return -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" on page 3-154 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 `sqrt` functions return the positive square root of the argument `x`.

Error Conditions

The `sqrt` functions return a zero if the input argument is negative.

Example

```
#include <math.h>
double y;
y = sqrt(2.0);      /* y = 1.414..... */
```

See Also

rsqrt

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

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

Example

```
#include <stdlib.h>
srand(22);
```

See Also

rand

sscanf

Convert formatted input in a string

Synopsis

```
#include <stdio.h>
int sscanf(const char *s, const char *format, /* args */...);
```

Description

The `sscanf` function reads from the string `s`. The function is equivalent to `fscanf` with the exception of the string being read from a string rather than a stream. The behavior of `sscanf` when reaching the end of the string equates to `fscanf` reaching the `EOF` in a stream. For details on the control format string, refer to "fscanf" on page 3-169.

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.

Error Conditions

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

Example

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

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

See Also

strncat

strchr

Find first occurrence of character in string

Synopsis

```
#include <string.h>
char *strchr(const char *s1, int c);
```

Description

The `strchr` function returns a pointer to the first location in `s1` (null-terminated string) that contains the character `c`.

Error Conditions

The `strchr` function returns a null pointer if `c` is not part of the string.

Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strchr(ptr1, 'E');
       /* ptr2 points to the E in TESTING */
```

See Also

memchr, 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.

Error Conditions

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

Example

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

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

See Also

memcmp, strncmp

strcoll

Compare strings

Synopsis

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

Description

The `strcoll` function compares the string pointed to by `s1` with the string pointed to by `s2`. The comparison is based on the `LC_COLLATE` locale macro. Because only the C locale is defined in the Blackfin run-time environment, the `strcoll` function is identical to the `strcmp` function. The function returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the `s2` string is greater than the `s1` string, and a zero if the strings are the same.

Error Conditions

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

Example

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

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

See Also

strcmp, strncmp

strcpy

Copy from one string to another

Synopsis

```
#include <string.h>
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`.

Error Conditions

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

Example

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

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

See Also

memcpy, memmove, strncpy

strcspn

Length of character segment in one string but not the other

Synopsis

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

Description

The `strcspn` function returns the 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.

Error Conditions

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

Example

```
#include <string.h>
char *ptr1, *ptr2;
size_t len;

ptr1 = "Tried and Tested";
ptr2 = "aeiou";
len = strcspn (ptr1,ptr2);      /* len = 2 */
```

See Also

strlen, strspn

strerror

Get string containing error message

Synopsis

```
#include <string.h>
char *strerror(int errnum);
```

Description

The `strerror` function returns a pointer to a string containing an error message by mapping the number in `errnum` to that string.

Error Conditions

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

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 Table 3-34. The `strftime` function only supports the “C” locale, and this is reflected in the table.

Table 3-34. Conversion Specifiers Supported by `strftime`

Conversion Specifier	Transformation	ISO/IEC 9899
`%a`	Abbreviated weekday name	Yes
`%A`	Full weekday name	Yes
`%b`	Abbreviated month name	Yes
`%B`	Full month name	Yes

Table 3-34. Conversion Specifiers Supported by `strftime` (Cont'd)

Conversion Specifier	Transformation	ISO/IEC 9899
`%c`	Date and time presentation in the form of `DDD MMM dd hh:mm:ss yyyy`	Yes
`%C`	Century of the year	POSIX.2-1992 + ISO C99
`%d`	Day of the month (`01 - 31`)	Yes
`%D`	Date represented as `mm/dd/yy`	POSIX.2-1992 + ISO C99
`%e`	Day of the month, padded with a space character (cf `%d`)	POSIX.2-1992 + ISO C99
`%F`	Date represented as `yyyy-mm-dd`	POSIX.2-1992 + ISO C99
`%h`	Abbreviated name of the month (same as `%b`)	POSIX.2-1992 + ISO C99
`%H`	Hour of the day as a 24-hour clock (`00-23`)	Yes
`%I`	Hour of the day as a 12-hour clock (`00-12`)	Yes
`%j`	Day of the year (`001-366`)	Yes
`%k`	Hour of the day as a 24-hour clock padded with a space (` 0-23`)	No
`%l`	Hour of the day as a 12-hour clock padded with a space (`0-12`)	No
`%m`	Month of the year (`01-12`)	Yes
`%M`	Minute of the hour (`00-59`)	Yes
`%n`	Newline character	POSIX.2-1992 + ISO C99
`%p`	`AM` or `PM`	Yes
`%P`	`am` or `pm`	No
`%r`	Time presented as either `hh:mm:ss AM` or as `hh:mm:ss PM`	POSIX.2-1992 + ISO C99
`%R`	Time presented as `hh:mm`	POSIX.2-1992 + ISO C99
`%S`	Second of the minute (`00-61`)	Yes
`%t`	Tab character	POSIX.2-1992 + ISO C99

Table 3-34. Conversion Specifiers Supported by `strftime` (Cont'd)

Conversion Specifier	Transformation	ISO/IEC 9899
`%T`	Time formatted as `%H:%M:%S`	POSIX.2-1992 + ISO C99
`%U`	Week number of the year (week starts on Sunday) (`00-53`)	Yes
`%w`	Weekday as a decimal (`0-6`) (`0` if Sunday)	Yes
`%W`	Week number of the year (week starts on Sunday) (`00-53`)	Yes
`%x`	Date represented as `mm/dd/yy` (same as `%D`)	Yes
`%X`	Time represented as `hh:mm:ss`	Yes
`%y`	Year without the century (`00-99`)	Yes
`%Y`	Year with the century (`nnnn`)	Yes
`%Z`	Time zone name, or nothing if the name cannot be determined	Yes
`%%`	`%` character	Yes

The current implementation of `time.h` does not support time zones and, therefore, the `%Z` specifier does not generate any characters.

The `strftime` function returns the number of characters (not including the terminating null character) that have been written to `buf`.

Error Conditions

The `strftime` function returns zero if more than `buf_size` characters are required to process the format string. In this case, the contents of the array `buf` will be indeterminate.

Example

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

extern void
print_time(time_t tod)
{
   char tod_string[100];

   strftime(tod_string,
            100,
            "It is %M min and %S secs after %l o'clock (%p)",
            gmtime(&tod));
   puts(tod_string);
}
```

See Also

ctime, gmtime, localtime, mktime

strlen

String length

Synopsis

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

Description

The `strlen` function returns the length of the null-terminated string pointed to by `s1` (not including the terminating null character).

Error Conditions

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

Example

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

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

See Also

strcspn, strspn

strncat

Concatenate characters from one string to another

Synopsis

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

Description

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

Error Conditions

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

Example

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

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

See Also

strcat

strncmp

Compare characters in strings

Synopsis

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

Description

The `strncmp` function lexicographically 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.

Error Conditions

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

Example

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

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

See Also

memcmp, strcmp

strncpy

Copy characters from one string to another

Synopsis

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

Description

The `strncpy` function copies up to `n` characters of the null-terminated string 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.

Error Conditions

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

Example

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

strncpy(string1, "MOREFUN", 4);
                        /* MORE is copied into string1 */
string1[4] = '\0';      /* must null-terminate string1 */
```

See Also

memcpy, memmove, strcpy

strpbrk

Find character match in two strings

Synopsis

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

Description

The `strpbrk` function returns a pointer to the first character in `s1` that is also found in `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

Error Conditions

In the event that no character in `s1` matches any in `s2`, a null pointer is returned.

Example

```
#include <string.h>
char *ptr1, *ptr2, *ptr3;

ptr1 = "TESTING";
ptr2 = "SHOP"
ptr3 = strpbrk(ptr1, ptr2);
   /* ptr3 points to the S in TESTING */
```

See Also

strspn

strrchr

Find last occurrence of character in string

Synopsis

```
#include <string.h>
char *strrchr(const char *s1, int c);
```

Description

The `strrchr` function returns a pointer to the last occurrence of character `c` in the null-terminated input string `s1`.

Error Conditions

The `strrchr` function returns a null pointer if `c` is not found.

Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strrchr(ptr1, 'T');
   /* ptr2 points to the second T of TESTING */
```

See Also

memchr, strchr

strspn

Length of segment of characters in both strings

Synopsis

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

Description

The `strspn` function returns the length of the initial segment of `s1`, which consists entirely of characters in the string pointed to by `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

Error Conditions

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

Example

```
#include <string.h>
size_t len;
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = "ERST";
len = strspn(ptr1, ptr2);      /* len = 4 */
```

See Also

strcspn, strlen

strstr

Find string within string

Synopsis

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

Description

The `strstr` function returns a pointer to the first occurrence in the string of `s1` of the characters pointed to by `s2`. This excludes the terminating null character in `s1`.

Error Conditions

If the string is not found, `strstr` returns a null pointer. If `s2` points to a string of zero length, `s1` is returned.

Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strstr (ptr1, "E");
   /* ptr2 points to the E in TESTING */
```

See Also

strchr

strtod

Convert string to double

Synopsis

```
#include <stdlib.h>
double strtod (const char *nptr, char **endptr)
```

Description

The `strtod` function extracts a value from the string pointed to by `nptr`, and returns the value as a `double`. The `strtod` function expects `nptr` to point to a string that represents either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

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

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

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

The form of a hexadecimal floating-point number is:

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

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

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

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

The first character that does not fit either form of number stops the scan. If `endptr` is not NULL, a pointer to the character that stopped the scan is stored at the location pointed to by `endptr`. If no conversion can be performed, the value of `nptr` is stored at the location pointed to by `endptr`.

Error Conditions

The `strtod` function returns a zero if no conversion 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 the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

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

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

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

The form of a hexadecimal floating-point number is:

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

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

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

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

The first character that does not fit either form of number stops the scan. If `endptr` is not NULL, a pointer to the character that stopped the scan is stored at the location pointed to by `endptr`. If no conversion can be performed, the value of `nptr` is stored at the location pointed to by `endptr`.

Error Conditions

The `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>

fract strtofxr(const char *nptr, char **endptr);
accum strtofxk(const char *nptr, char **endptr);

short fract strtofxhr(const char *nptr, char **endptr);
short accum strtofxhk(const char *nptr, char **endptr);

long fract strtofxlr(const char *nptr, char **endptr);
long accum strtofxlk(const char *nptr, char **endptr);

unsigned fract strtofxur(const char *nptr, char **endptr);
unsigned accum strtofxuk(const char *nptr, char **endptr);

unsigned short fract strtofxuhr(const char *nptr, char **endptr);
unsigned short accum strtofxuhk(const char *nptr, char **endptr);

unsigned long fract strtofxulr(const char *nptr, char **endptr);
unsigned long accum strtofxulk(const char *nptr, char **endptr);
```

Description

The `strtofx`*`fx`* family of functions extracts a value from the string pointed to by `nptr`, and converts the value to a fixed-point representation. The `strtofx`*`fx`* functions expect `nptr` to point to a string that represents either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

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

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

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

The form of a hexadecimal floating-point number is:

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

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

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

The first character that does not fit either form of number stops the scan. If `endptr` is not NULL, a pointer to the character that stopped the scan is stored at the location pointed to by `endptr`. If no conversion can be performed, the value of `nptr` is stored at the location pointed to by `endptr`.

Error Conditions

The `strtofx`*`fx`* functions return a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`. If the correct value results in an overflow, the maximum positive or negative (as appropriate) fixed-point value is returned. If the correct value results in an underflow, zero is returned. The `ERANGE` value is stored in `errno` in the case of overflow.

Example

```
#include <stdfix.h>
char *rem;
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 whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

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

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

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

The form of a hexadecimal floating-point number is:

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

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

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

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

The first character that does not fit either form of number stops the scan. If `endptr` is not NULL, a pointer to the character that stopped the scan is stored at the location pointed to by `endptr`. If no conversion can be performed, the value of `nptr` is stored at the location pointed to by `endptr`.

Error Conditions

The `strtold` function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`. If the correct value results in an overflow, a positive or negative (as appropriate) `LDBL_MAX` is returned. If the correct value results in an underflow, zero is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

Example

```
#include <stdlib.h>
char *rem;
long double dd;

dd = strtold ("2345.5E4 abc",&rem);
     /* dd = 2.3455E+7, rem = " abc" */

dd = strtold ("-0x1.800p+9,123",&rem);
     /* dd = -768.0, rem = ",123"   */
```

See Also

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 `unsigned long int` the value represented by the string `nptr`. If `endptr` is not a null pointer, `strtoul` stores a pointer to the unconverted remainder in `*endptr`.

The `strtoul` function breaks down the input into three sections:

- 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.

Error Conditions

The strxfrm function does not return an error condition.

Example

```
#include <string.h>
char string1[50];
strxfrm(string1, "SOMEFUN", 49);
      /* SOMEFUN is copied into string1 */
```

See Also

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 `tan` functions return the tangent of `x`. Both the argument `x` and the function results are in radians. The defined domain for the `tanf` function is [-9099, 9099], and for the `tand` function the domain is [-4.216e8, 4.216e8].

The `tan_fr16`, `tan_fr32`, `tan_fx16` and `tan_fx32` functions are defined for fractional input values between [- π/4, π/4]. The outputs from the functions are in the range [-1.0, 1.0).

Error Conditions

The `tan` 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 `tanh` functions return the hyperbolic tangent of the argument `x`, where `x` is measured in radians.

Error Conditions

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

Example

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

y = tanh (x);
z = tanhf (w);
```

See Also

cosh, sinh

time

Calendar time

Synopsis

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

Description

The `time` function returns the current calendar time, which measures the number of seconds that have elapsed since the start of a known epoch. As the calendar time cannot be determined in this implementation of `time.h`, a result of `(time_t)-1` is returned. The function result is also assigned to its argument, if the pointer to `t` is not a null pointer.

Error Conditions

The `time` function will return the value `(time_t) -1` if the calendar time is not available.

Example

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

if (time(NULL) == (time_t) -1)
    printf("Calendar time is not available\n");
```

See Also

ctime, gmtime, localtime

tmpfile

Create a temporary file

Synopsis

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

Description

This function is not thread-safe, and is only available 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 file will be deleted when it is closed or when the application terminates. Note that the file is deleted via the `remove` function, which is only supported under the default device driver supplied by the VisualDSP++ simulator and EZ-KIT Lite evaluation system, and only operates on the host file system.

If successful, the function will return a pointer to the stream; if the function could not open a temporary file, it will return NULL.

> 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 will return 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 will assume that the pointer is to an array of at least `L_TMPNAM` characters, and it will copy 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.

> The file name generated is a valid file name that is not the same as the name of an existing file. This implementation will ensure that it is unique by calling the `remove` function to delete any existing version of the file. Note that the `remove` function is only supported under the default device driver supplied by the VisualDSP++ simulator and EZ-KIT Lite evaluation system, and it only operates on the host file system.

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 will generate 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 will return 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

The `tmpnam` function does not return any errors.

Example

```
#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

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

Description

The `tolower` function converts the input character to lowercase if it is uppercase; otherwise, it returns the character.

Error Conditions

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

Example

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

for (ch=0; ch<=0x7f; ch++) {
   printf("%#04x", ch);
   if(isupper(ch))
      printf("tolower=%#04x", tolower(ch));
   putchar('\n');
}
```

See Also

islower, isupper, toupper

toupper

Convert from lowercase to uppercase

Synopsis

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

Description

The `toupper` function converts the input character to uppercase if it is in lowercase; otherwise, it returns the character.

Error Conditions

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

Example

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

for (ch=0; ch<=0x7f; ch++) {
   printf("%#04x", ch);
   if(islower(ch))
      printf("toupper=%#04x", toupper(ch));
   putchar('\n');
}
```

See Also

islower, isupper, tolower

ungetc

Push character back into input stream

Synopsis

```
#include <stdio.h>
int ungetc(int uc, FILE *stream);
```

Description

The `ungetc` function pushes the character specified by `uc` back onto `stream`. The characters that have been pushed back onto `stream` will be returned by any subsequent read of `stream` in the reverse order of their pushing.

A successful call to the `ungetc` function will clear the `EOF` indicator for `stream`. The file position indicator for `stream` is decremented for every successful call to `ungetc`.

Upon successful completion, `ungetc` returns the character pushed back after conversion.

Error Conditions

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

Example

```
#include <stdio.h>

void ungetc_example(FILE *fp)
{
   int ch, ret_ch;
   /* get char from file pointer */
   ch = fgetc(fp);
   /* unget the char, return value should be char */
```

```
    if ((ret_ch = ungetc(ch, fp)) != ch)
       printf("ungetc failed\n");
    /* make sure that the char had been placed in the file */
    if ((ret_ch = fgetc(fp)) != ch)
       printf("ungetc failed to put back the char\n");
}
```

See Also

fseek, fsetpos, getc

va_arg

Get next argument in variable-length list of arguments

Synopsis

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

Description

The `va_arg` macro is used to walk through the variable-length list of arguments to a function.

After starting to process a variable-length list of arguments with `va_start`, call `va_arg` with the same `va_list` variable to extract arguments from the list. Each call to `va_arg` returns a new argument from the list.

Substitute a `type` name corresponding to the type of the next argument for the `type` parameter in each call to `va_arg`. After processing the list, call `va_end`.

The `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

The va_arg macro does not return an error condition.

Example

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

char *concat(char *s1,...)
{
   int len = 0;
   char *result;
   char *s;
   va_list ap;

   va_start (ap,s1);
   s = s1;
   while (s){
      len += strlen (s);
      s = va_arg (ap,char *);
   }
   va_end (ap);

   result = malloc (len +7);
   if (!result)
      return result;
   *result = '\0';
   va_start (ap,s1);
   s = s1;
```

```
    while (s){
        strcat (result,s);
        s = va_arg (ap,char *);
    }
    va_end (ap);
    return result;
}

char *txt1 = "One";
char *txt2 = "Two";
char *txt3 = "Three";

extern int main(void)
{
    char *result;

    result = concat(txt1, txt2, txt3, NULL);

    puts(result);   /* prints "OneTwoThree" */
    free(result);
}
```

See Also

va_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

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

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 is used to start processing variable arguments in a function declared to take a variable number of arguments. The first argument to `va_start` should be a variable of type `va_list`, which is used by `va_arg` to walk through the arguments.

The second argument is the name of the last *named* parameter in the function's parameter list; the list of variable arguments immediately follows this parameter. The `va_start` macro must be invoked before either the `va_arg` or `va_end` macro can be invoked.

Error Conditions

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

See Also

va_arg, va_end

vfprintf

Print formatted output of a variable argument list

Synopsis

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

int vfprintf(FILE *stream, const char *format, va_list ap);
```

Description

The `vfprintf` function formats data according to the argument `format`, and then writes the output to the stream `stream`. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to "fprintf" on page 3-154 for a description of the valid format specifiers.

The `vfprintf` function behaves in the same manner as `fprintf` with the exception that instead of being a function which takes a variable number or arguments it is called with an argument list `ap` of type `va_list`, as defined in `stdarg.h`.

If the `vfprintf` function is successful it will return the number of characters output.

Error Conditions

The `vfprintf` function returns a negative value if unsuccessful.

Example

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

void write_name_to_file(FILE *fp, char *name_template, ...)
```

```
{
   va_list p_vargs;
   int ret;                   /* return value from vfprintf */

   va_start (p_vargs,name_template);
   ret = vfprintf(fp, name_template, p_vargs);
   va_end (p_vargs);

   if (ret < 0)
      printf("vfprintf failed\n");
}
```

See Also

fprintf, va_start, va_end

vprintf

Print formatted output of a variable argument list to stdout

Synopsis

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

int vprintf(const char *format, va_list ap);
```

Description

The vprintf function formats data according to the argument format, and then writes the output to the standard output stream stdout. The argument format contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to "fprintf" on page 3-154 for a description of the valid format specifiers.

The vprintf function behaves in the same manner as vfprintf with stdout provided as the pointer to the stream.

If the vprintf function is successful it will return the number of characters output.

Error Conditions

The vprintf function returns a negative value if unsuccessful.

Example

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

void print_message(int error, char *format, ...)
```

```
{
   /* This function is called with the same arguments as for */
   /* printf but if the argument error is not zero, then the */
   /* output will be preceded by the text "ERROR:"           */

   va_list p_vargs;
   int ret;                   /* return value from vprintf */

   va_start (p_vargs, format);
   if (!error)
      printf("ERROR: ");
   ret = vprintf(format, p_vargs);
   va_end (p_vargs);

   if (ret < 0)
      printf("vprintf failed\n");
}
```

See Also

fprintf, vfprintf

vsnprintf

Format argument list into an n-character array

Synopsis

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

int vsnprintf (char *str, size_t n, const char *format,
               va_list args);
```

Description

The `vsnprintf` function is similar to the `vsprintf` function in that it formats the variable argument list `args` according to the argument `format`, and then writes the output to the array `str`. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to "fprintf" on page 3-154 for a description of the valid format specifiers.

The function differs from `vsprintf` in that no more than `n-1` characters are written to the output array. Any data written beyond the `n-1'th` character is discarded. A terminating `NUL` character is written after the end of the last character written to the output array unless `n` is set to zero, in which case nothing will be written to the output array and the output array may be represented by the `NULL` pointer.

The `vsnprintf` function returns the number of characters that would have been written to the output array `str` if `n` was sufficiently large. The return value does not include the terminating `NUL` character written to the array.

Error Conditions

The `vsnprintf` function returns a negative value if unsuccessful.

Example

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

char *message(char *format, ...)
{
   char *message = NULL;
   int len = 0;
   int r;
   va_list p_vargs;        /* return value from vsnprintf  */

   do {
      va_start (p_vargs,format);
      r = vsnprintf (message,len,format,p_vargs);
      va_end (p_vargs);
      if (r < 0)           /* formatting error?            */
         abort();
      if (r < len)         /* was complete string written? */
         return message;   /* return with success          */
      message = realloc (message,(len=r+1));
   } while (message != NULL);
   abort();
}
```

See Also

fprintf, snprintf

vsprintf

Format argument list into a character array

Synopsis

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

int vsprintf (char *str, const char *format, va_list args);
```

Description

The `vsprintf` function formats the variable argument list `args` according to the argument `format`, and then writes the output to the array `str`. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to "fprintf" on page 3-154 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 or arguments function it is called with an argument list `args` of type `va_list`, as defined in `stdarg.h`.

The `vsprintf` function returns the number of characters that have been written to the output array `str`. The return value does not include the terminating `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

4 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 Chapter 3, “C/C++ Run-Time Library”.

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.

In addition to containing the user-callable functions described in this chapter, the DSP run-time library also contains compiler support functions that perform basic operations on integer and floating-point types that the compiler might not perform in-line. These functions are called by compiler-generated code to implement basic type conversions, floating-point operations, and so on. Compiler support functions should not be called directly from user code.

This chapter contains:

- "DSP Run-Time Library Guide" on page 4-2
  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" on page 4-75
  contains the complete reference for each DSP run-time library function provided with this release of the `ccblkfn` compiler.

DSP Run-Time Library Guide

The DSP run-time library contains functions that can be called from your source program. This section includes:

Linking DSP Library Functions

The DSP run-time library is located under the VisualDSP++ installation directory in the subdirectory `Blackfin/lib`. Different versions of the library are supplied and catalogued in Table 4-1.

Table 4-1. DSP Library Files

Blackfin/lib Directory	Description

`libdsp532.dlb`
`libdsp535.dlb`
`libdsp561.dlb`
	DSP run-time library

`libdsp532y,dlb`
`libdsp535y.dlb`
`libdsp561y.dlb`
	DSP run-time library built with the `-si-revision` flag specified. (For more information, see "-si-revision" on page 1-74.)

Versions of the DSP run-time library that contain "`532`" in the file name have been built to run on ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF534, ADSP-BF536, ADSP-BF537, ADSP-BF538, or ADSP-BF539 processors. Versions of the DSP run-time library that contain "`535`" in the file name have been built to run on ADSP-BF535 processors. Versions of the DSP run-time library that contain "`561`" in the file name have been built to run on ADSP-BF561 processors.

Versions of the library whose file name end with a "`y`" (for example, `libdsp532y.dlb`) are built with the compiler's `-si-revision` switch and include all available compiler workarounds for hardware anomalies. (See "-si-revision" on page 1-74.)

When an application calls a DSP library function, the call creates a reference that the linker resolves. One way to direct the linker to the library's location is to use the default linker description file (<*your_target*>`.ldf`). If a customized `.ldf` file is used to link the application, add the appropriate DSP run-time library to the `.ldf` file used by the project.

> Instead of modifying a customized `.ldf` file, use the `-l` switch (see “-l” on page 1-47) to specify the library that should be searched by the linker. For example, the `-ldsp532` switch adds the `libdsp532.dlb` library to the list of libraries that the linker examines. For information on `.ldf` files, refer to the *VisualDSP++ Linker and Utilities Manual.*

Working With Library Source Code

The source code for the functions in the DSP run-time library is provided with VisualDSP++. By default, the libraries are installed in the directory `Blackfin/lib`, and the source files are copied into `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” on page 1-408 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.

> 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” in Chapter 3, C/C++ Run-Time Library.

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. Table 4-2 shows the DSP header files included in this release of the `ccblkfn` compiler.

Table 4-2. DSP Header Files

Header File	Description
`complex.h`	Basic complex arithmetic functions (on page 4-5)
`cycle_count.h`	Basic cycle counting (on page 4-9)
`cycles.h`	Cycle counting with statistics (on page 4-10)
`filter.h`	Filters and transformations (on page 4-10)
`math.h`	Math functions (on page 4-20)
`matrix.h`	Matrix functions (on page 4-24)
`stats.h`	Statistical functions (on page 4-38)
`vector.h`	Vector functions (on page 4-45)
`window.h`	Window generators (on page 4-61)

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`.

The complex functions defined in this header file are listed in Table 4-3 on page 4-7. Functions that operate on 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;
```

Details about basic complex arithmetic functions are included in "DSP Run-Time Library Reference" starting on page 4-75.

Table 4-3. Complex Functions

Description	Prototype

Complex Absolute Value	`double cabs (complex_double a)`
	`float cabsf (complex_float 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)`

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)`

Table 4-3. Complex Functions (Cont'd)

Description	Prototype

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)`

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)`
	`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_fract16 conj_fr16 (complex_fract16 a)`
	`complex_fract32 conj_fr32 (complex_fract32 a)`

Table 4-3. Complex Functions (Cont'd)

Description	Prototype

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)`
	`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)`

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" on page 4-64.

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" on page 4-64.

filter.h

The `filter.h` header file contains 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 functions that perform key signal processing transformations, including FFTs and convolution.

The library provides various forms of the FFT function, corresponding to radix-2, radix-4, and two-dimensional FFTs. The number of points is provided as an argument. The header file also defines a complex FFT function (`cfftf_fr16`) implemented using an optimized radix-4 algorithm. However, the `cfftf_fr16` function has certain requirements that may not be appropriate for some applications. The twiddle table for the FFT functions is supplied as a separate argument and is normally calculated once during program initialization.

The `cfftf_fr16` library function uses the `M3` register, which may be used by an emulator for context switching. Refer to the appropriate emulator documentation.

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.

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 Table 4-4 and Table 4-5 and are described in "DSP Run-Time Library Reference" on page 4-75.

Table 4-4. 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, iirdf1_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)`

Table 4-4. Filter Library (Cont'd)

Description	Prototype

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[],`
	`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)`

Table 4-4. Filter Library (Cont'd)

Description	Prototype
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 4-5. Transformational Functions

Description	Prototype
Fast Fourier Transforms	
Generate FFT Twiddle Factors	`void twidfft_fr16 (complex_fract16 twiddle_table[], int fft_size)`
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 Radix-4 FFT	`void twidfftrad4_fr16 (complex_fract16 twiddle_table[], int fft_size)`

Table 4-5. Transformational Functions (Cont'd)

Description	Prototype
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)`
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)`

Table 4-5. Transformational Functions (Cont'd)

Description	Prototype

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, 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)`
N Point Radix-4 Complex Input FFT	`void cfftrad4_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)`

Table 4-5. Transformational Functions (Cont'd)

Description	Prototype

N Point Radix-4
Real Input FFT
	`void rfftrad4_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)`

N Point Radix-4
Inverse Input FFT
	`void ifftrad4_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)`

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)`

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,`
	`     complex_fract32 *temp, complex_fract32 *output,`
	`     const complex_fract32 *twiddle_table,`
	`     int twiddle_stride, int fft_size)`

Table 4-5. Transformational Functions (Cont'd)

Description	Prototype

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 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)

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)

Fast N point Mixed-Radix
Complex Input FFT
	void cfftf_fr32
	    (const complex_fract32 *input,
	     complex_fract32 *output,
	     const complex_fract32 *twiddle_table,
	     int twiddle_stride, int fft_size)

Table 4-5. Transformational Functions (Cont'd)

Description	Prototype

Fast N point Mixed-Radix Inverse Input FFT	`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_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)`

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,`
	`     const long _Fract input_y[], int length_y,`
	`     long _Fract output[])`

Table 4-5. Transformational Functions (Cont'd)

Description	Prototype

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)`

Table 4-5. Transformational Functions (Cont'd)

Description	Prototype
μ-law compression	`void mu_compress (const short input[], short output[], int length)`
μ-law expansion	`void mu_expand (const char 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`.

Table 4-6 summarizes the functions defined by the `math.h` header file. Descriptions of these functions are given under the name of the `double` version in "C Run-Time Library Reference" on page 3-64.

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" on page 4-75.

Table 4-6. 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)`

Table 4-6. Math Library (Cont'd)

Description	Prototype

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)`

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

Table 4-6. Math Library (Cont'd)

Description	Prototype

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)`

Table 4-6. Math Library (Cont'd)

Description	Prototype

Natural Logarithm	`double log (double x)`
	`float logf (float x)`
	`long double logd (long double x)`

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)`

Table 4-6. Math Library (Cont'd)

Description	Prototype

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" on page 4-5 for definitions of the complex types.

The matrix functions defined in the `matrix.h` header file are listed in Table 4-7. 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 4-7. 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)`

Table 4-7. Matrix Functions (Cont'd)

Description	Prototype

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)`

Table 4-7. Matrix Functions (Cont'd)

Description	Prototype

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 4-7. Matrix Functions (Cont'd)

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)`

Table 4-7. Matrix Functions (Cont'd)

Description	Prototype

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)`

Table 4-7. Matrix Functions (Cont'd)

Description	Prototype

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 4-7. Matrix Functions (Cont'd)

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)`

Table 4-7. Matrix Functions (Cont'd)

Description	Prototype

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)`

Table 4-7. Matrix Functions (Cont'd)

Description	Prototype

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)`

Table 4-7. Matrix Functions (Cont'd)

Description	Prototype

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, 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)`

Table 4-7. Matrix Functions (Cont'd)

Description	Prototype

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)`

Table 4-7. Matrix Functions (Cont'd)

Description	Prototype

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)`

Table 4-7. Matrix Functions (Cont'd)

Description	Prototype

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)`

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 Table 4-8 and are described in "DSP Run-Time Library Reference" on page 4-75.

Table 4-8. 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[])`
	`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[])`

Table 4-8. Statistical Functions (Cont'd)

Description	Prototype

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[])`

Table 4-8. Statistical Functions (Cont'd)

Description	Prototype

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[])`

Table 4-8. Statistical Functions (Cont'd)

Description	Prototype

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[])`
	`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[])`

Table 4-8. Statistical Functions (Cont'd)

Description	Prototype

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)`

Table 4-8. Statistical Functions (Cont'd)

Description	Prototype

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)`

Table 4-8. Statistical Functions (Cont'd)

Description	Prototype

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)`
	`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" on page 4-5 for definitions of the complex types.

The functions defined in the `vector.h` header file are listed in Table 4-9. 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 4-9. 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 4-9. Vector Functions (Cont'd)

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)`

Table 4-9. Vector Functions (Cont'd)

Description	Prototype

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)`

Table 4-9. Vector Functions (Cont'd)

Description	Prototype

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 4-9. Vector Functions (Cont'd)

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)`

Table 4-9. Vector Functions (Cont'd)

Description	Prototype

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)`

Table 4-9. Vector Functions (Cont'd)

Description	Prototype

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)`
	`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)`

Table 4-9. Vector Functions (Cont'd)

<table>
<tr><th>Description</th><th>Prototype</th></tr>
<tr><td>Complex Vector +
Scalar Addition</td><td><pre>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)</pre></td></tr>
</table>

Table 4-9. Vector Functions (Cont'd)

Description	Prototype

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)`

Table 4-9. Vector Functions (Cont'd)

Description	Prototype

Complex Vector * Scalar Multiplication	`void cvecsmlt (const complex_double vec[], complex_double scalar, 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)`

Table 4-9. Vector Functions (Cont'd)

Description	Prototype

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)`

Table 4-9. Vector Functions (Cont'd)

Description	Prototype

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)`

Table 4-9. Vector Functions (Cont'd)

Description	Prototype

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[], 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)`

Table 4-9. Vector Functions (Cont'd)

Description	Prototype

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 Table 4-10 and are described in "DSP Run-Time Library Reference" on page 4-75.

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 4-10. 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)`

Table 4-10. Window Generator Functions (Cont'd)

Description	Prototype

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)`
	`void gen_hanning_fx32 (long _Fract hanning_window[], int window_stride, int window_size)`

Table 4-10. Window Generator Functions (Cont'd)

Description	Prototype

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)`

Table 4-10. Window Generator Functions (Cont'd)

Description	Prototype

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.

> 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" on page 4-65
- "Cycle-Counting Facility With Statistics" on page 4-67
- "Using time.h to Measure Cycle Counts" on page 4-70
- "Determining the Processor Clock Rate" on page 4-72
- "Considerations When Measuring Cycle Counts" on page 4-73

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;
```

> 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 macro `PRINT_CYCLES(STRING,T)` which is provided mainly as an example of how to print a value of type `cycle_t`; the macro outputs the text `STRING` 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" on page 4-67 and "Using time.h to Measure Cycle Counts" on page 4-70); the relative benefits of this facility are outlined in "Considerations When Measuring Cycle Counts" on page 4-73.

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 recompilation 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 might output:

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

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

Alternative methods of measuring the performance of compiled C source are described in "Basic Cycle-Counting Facility" on page 4-65 and "Using time.h to Measure Cycle Counts" on page 4-70. Also refer to "Considerations When Measuring Cycle Counts" on page 4-73, 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" on page 4-72).
- 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" on page 4-65 and "Cycle-Counting Facility With Statistics" on page 4-67, respectively. Also refer to "Considerations When Measuring Cycle Counts" on page 4-73, 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=600000000LL`).

- By way of the System Services Library

- By way of the **Processor speed** box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Processor (1)** category

- 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" on page 4-65
  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" on page 4-67
  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" on page 4-70
  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.

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, formatted as follows:

Name and purpose of the function

Synopsis – Required header file and functional prototype; when the functionality is provided for several data types (for example, `float`, `double`, `long double`, or `fract16`), several prototypes are given

Description – Function specification

Algorithm – High-level mathematical representation of the function

Domain – Range of values supported by the function

Notes – Miscellaneous information

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`.

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 is expanded to a linear 13-bit signed sample in accordance with the A-law definition and is returned in the vector pointed to by `output`.

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's domain. For input values less than the domain, the functions return 0.0.

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.

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, log10, pow

arg

Get phase of a complex number

Synopsis

```
#include <complex.h>

float argf (complex_float a);
double arg (complex_double a);
long double argd (complex_long_double a);

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.

Refer to the description of the `polar_fr16` function (see "polar" on page 4-222), which explains how a phase, represented as a fractional number, is interpreted in polar notation.

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 ( )`

autocoh

Autocoherence

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 autocoherence of the input vector `samples[]`, which contain `sample_length` values. The autocoherence of an input signal is its autocorrelation minus its mean squared. The functions return the result in the output array `coherence[]` of length `lags`.

Algorithm

The following equation is the basis of the algorithm.

$$c_k = \frac{1}{n} \sum_{j=0}^{n-k-1} (a_j \bullet a_{j+k}) - (\bar{a})^2$$

where:

k = {0, 1, ..., `lags`-1}

$\bar{a}$ is the mean value of input vector `a`

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( )`

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 `autocorr` 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.

Algorithm

The following equation is the basis of the algorithm.

$$c_k = \frac{1}{n} \sum_{j=0}^{n-k-1} a_j \bullet a_{j+k}$$

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 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.

Algorithm

The following equation is the basis of the algorithm.

$$c = \sqrt{Re^2(a) + Im^2(a)}$$

Domain

$Re^2(a) + Im^2(a) <= 3.4 \times 10^{38}$ for `cabsf( )`

$Re^2(a) + Im^2(a) <= 1.7 \times 10^{308}$	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.

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`.

Refer to the description of the `polar_fr16` function (see "polar" on page 4-222), which explains how a phase, represented as a fractional number, is interpreted in polar notation.

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.

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( )` for `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.

Algorithm

```
Re(c) = cos(x)
Im(c) = sin(x)
```

Domain

x = [–102940 , 102940]	for `cexpf( )`
x = [-8.433e8 , 8.433e8]	for `cexpd( )`

cfft

N-point radix-2 complex input FFT

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 (on page 4-242) for cfft_fr16 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 will 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 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.

Any values for the argument `scale_method` other than 2 or 3 will result in the function performing static scaling.

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 radix-4 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 `cfftf_fr16` function "decimates in frequency" using an optimized radix-4 algorithm, with the `cfftf_fr32` function 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 `cfftf_fr16` function has been designed for optimal performance and requires that the input array `input` be aligned on an address boundary that is a multiple of four times the FFT size. For certain applications, this alignment constraint may not be appropriate; in such cases, the application should call the `cfft_fr16` function (on page 4-98) instead, with no loss of facility (apart from performance).

The number of points in the FFT (`fft_size`) must be a power of 4 and must be at least 16 for `cfftf_fr16` and a power of 2 and at least 8 for `cfftf_fr32`.

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` (on page 4-247) may be used to initialize the array for `cfftf_fr16` with `twidfftf_fr32` (on page 4-247) used to initialize the array for `cfftf_fr32`.

If the twiddle table has been generated for an FFT of size `fft_size`, 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 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`.

The `cfftf_fr16` function uses the `M3` register, which may be used by an emulator for context switching. Refer to the appropriate emulator documentation.

Algorithm

The following equation is the basis of the algorithm.

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}$$

The `cfft_fr16` function (on page 4-98), which uses a radix-2 algorithm, must be used when the FFT size (`n`) is only a power of 2. The `cfftf_fr32` function uses a mixed-radix algorithm (radix-4 and radix-2).

Domain

For the `cfft_fr16` function, the number of points in the FFT must be a power of 4 and must be at least 16.

For the `cfft_fr32` function, the number of points in the FFT must be a power of 2 and must be at least 8.

Example

```
#include <filter.h>

#define FFTSIZE 64

#pragma align 256
segment ("seg_1") complex_fract16 input[FFTSIZE];

#pragma align 4
segment ("seg_2") complex_fract16 output[FFTSIZE];
```

```
#pragma align 4
segment ("seg_3") complex_fract16 twid[(3*FFTSIZE)/4];

twidfftf_fr16(twid,FFTSIZE);
cfftf_fr16(input,
           output,
           twid,1,FFTSIZE);
```

cfftrad4

N-point radix-4 complex input FFT

Synopsis

```
#include <filter.h>

void cfftrad4_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);
```

Description

This function transforms the time domain complex input signal sequence to the frequency domain by using the radix-4 Fast Fourier Transform. The `cfftrad4_fr16` function "decimates in frequency" by the radix-4 FFT algorithm.

The size of the input array `input`, the output array `output`, and the temporary working buffer `temp` is `fft_size`, where `fft_size` represents the number of points in the FFT. Memory bank collisions, which have an adverse effect on run-time performance, may be avoided by allocating all input and working buffers to different memory banks. If the input data can be overwritten, the 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 `3*fft_size/4` twiddle coefficients. The function `twidfftrad4_fr16` may be used to initialize the array. If the twiddle table contains more coefficients than needed for a particular call on cfftrad4_fr16, the stride factor must be set appropriately; otherwise it should be set to 1.

This function is provided for backward compatibility with existing applications. New applications should use the cfft_fr16 function (on page 4-98) instead.

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 fft_size must be a power of 4 and at least 16.

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 stores 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 number of points in the FFT must be a power of 2 and must be at least 4 for `cfft2d_fr16` and 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, 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` 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` (on page 4-250) 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.

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_fr32`.

Algorithm

The following equation is the basis of the algorithm.

$$c(i,j) = \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} a(k,l) \bullet e^{(-2\pi \cdot (i \cdot k + j \cdot l))/n}$$

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` and at least 8 for `cfft2d_fr32`.

Example

```
#include <filter.h>
#define FFT_SIZE1         32
#define FFT_SIZE2          8
#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 function. The macro `cfir_init`, in the `filter.h` header file, is available to initialize the structure.

It is defined as:

```
#define cfir_init(state, coeffs, delay, ncoeffs) \
   (state).h = (coeffs);  \
   (state).d = (delay);   \
   (state).p = (delay);   \
   (state).k = (ncoeffs)
```

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.

Error Conditions

The `cfir` functions check that the number of samples and the number of coefficients are positive - if not, the functions just returns.

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.

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.

Error Conditions

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

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 on page 4-209), 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.

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 functions 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.

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 data type, and this may affect the performance of the user-designed filter.

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 , $-a_0 1$, $b_0 1$, $-a_0 2$, $b_0 2$, ..., $-a_n 1$, $b_n 1$, $-a_n 2$, $b_n 2$, scale factor]

where n is the number of stages.

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 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.

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`.

Algorithm

Convolution between two sequences `input_x` and `input_y` is described as:

$$cout[n] = \sum_{k=0}^{clen2-1} cin1[n + k - (clen2 - 1)] \bullet cin2[(clen2 - 1) - k]$$

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`).

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.

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:

$$output[r][c] = \sum_{i=0}^{rows\_x-1} \sum_{k=0}^{cols\_x-1} x[j][k] \bullet y[r-j][c-k]$$

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 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]`.

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:

$$output[r][c] = \sum_{j=0}^{2} \sum_{k=0}^{2} x[(rows\_x+m)\%rows\_x][cols\_x+n)\%cols\_x] \bullet y[j][k]$$

where:

`r = 0` to `rows_x - 1`
`c = 0` to `cols_x - 1`
`m = r+j-1`
`n = c+k-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.

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.

Algorithm

```
c = cot(a)
```

Domain

a = [–9099 , 9099]	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`.

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 j^{th} 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_fx16 (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 cross-coherance functions compute the cross-coherence of two input vectors `samples_x[]` and `samples_y[]`. The cross-coherence is the cross-correlation minus the product of the mean of `samples_x` and the mean of `samples_y`. The length of the input vectors is given by `sample_length`. The functions return the result in the array `coherence` with `lags` elements.

Algorithm

The following equation is the basis of the algorithm.

$$C_k = \frac{1}{n} \bullet \sum_{j=0}^{n-k-1} (a_j \bullet b_{j+k}) - (\bar{a} \bullet \bar{b})$$

where:

k = {0, 1, ..., `lags`-1}
`a` = `samples_x`
`b` = `samples_y`
c = `coherence`
$\bar{a}$ is the mean value of input vector `a`
$\bar{b}$ is the mean value of input vector `b`

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( )`

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.

Algorithm

The following equation is the basis of the algorithm.

$$c_k = \frac{1}{n} \bullet \sum_{j=0}^{n-k-1} a_j \bullet b_{j+k}$$

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.

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 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, cfftrad4_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, rfftrad4_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` or `cfftf_fr32`, `cfftrad4_fr16`, `rfftrad4_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 will be in the range `0` to `log2(fft_size)`).

> 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)`.

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):

$$\text{fft\_magnitude[i]} = \frac{\text{sqrt}(\text{input[i].re}^2 + \text{input[i].im}^2)}{\text{fft\_size}}$$

where: `i = [0 ... fft_size)`

For mode 1 (rfft generated input):

$$\text{fft\_magnitude[i]} = \frac{2 * (\text{sqrt}(\text{input[i].re}^2 + \text{input[i].im}^2))}{\text{fft\_size}}$$

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) \
   (state).h = (coeffs);   \
   (state).d = (delay);    \
   (state).p = (delay);    \
   (state).k = (ncoeffs);  \
   (state).l = (index)
```

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.

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. See "#pragma align num" on page 1-280 for more information.

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 the fir functions. This field is normally set to an interpolation/decimation index before calling either the `fir_interp` or `fir_decima` functions.

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.

$$y(i) = \sum_{j=0}^{k-1} h(j) \bullet x(i-j)$$

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 NUM_SAMPLES   256
#define NUM_COEFFS     89

fract32 input[NUM_SAMPLES];
fract32 output[NUM_SAMPLES];
section("L1_data_a") fract32 coeffs[NUM_COEFFS];
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.

```
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) \
   (state).h = (coeffs);   \
   (state).d = (delay);    \
   (state).p = (delay);    \
   (state).k = (ncoeffs);  \
   (state).l = (index)
```

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.

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];
section("L1_data_a") fract32 coeffs[NUM_COEFFS];
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 performs 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` function 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) \
   (state).h = (coeffs);  \
   (state).d = (delay);   \
   (state).p = (delay);   \
```

```
(state).k = (ncoeffs); \
(state).l = (index)
```

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.

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.

$$y(i \bullet l + m) = \sum_{j=0}^{k-1} x(i-j) \bullet h(m \bullet k + j)$$

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

```
#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)
*/
```

```
section("L1_data_a") fract16 coeffs[N_COEFFS];
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_fr32(fract32     bartlett_window[],
                       int         window_stride,
                       int         window_size);

void gen_bartlett_fx16(_Fract      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 (on page 4-183) 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.

Algorithm

The following equation is the basis of the algorithm.

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

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`.

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.

Algorithm

The following equation is the basis of the algorithm.

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

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
$\alpha > 0$

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`.

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.

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(fract32      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.

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`.

Algorithm

The following equation is the basis of the algorithm.

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

where:

N	`window_size`
w	`kaiser_window`
n	{0, 1, 2, ..., N-1}
α	(N - 1) / 2
$I_0(\beta)$	Zeroth-order 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_rectangular` 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`.

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 (on page 4-166) regarding the relationship between it and the triangle window.

Algorithm

For even n, the following equation applies.

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

where:

N = window_size
w = triangle_window
n = {0, 1, 2, ..., N-1}

For odd n, the following equation applies.

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

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

Domain

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 (see on page 4-175).

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[ ]` that 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.

> ⓘ 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.

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

N-point radix-2 inverse FFT

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.

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.

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 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 will 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 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.

Any values for the argument `scale_method` other than 2 or 3 will result in the function performing static scaling.

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.

$$x(n) = \sum_{k=0}^{N-1} X(k)\, W_N^{-nk}$$

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

Inverse fast N-point Fast Fourier Transform

Synopsis

```
#include <filter.h>

void ifftf_fr32(const complex_fract32 input[],
                complex_fract32       output[],
                const complex_fract32 twiddle_table[],
                int                   twiddle_stride,
                int                   fft_size);
```

Description

The `ifftf_fr32` function transforms the frequency domain complex input signal sequence to the time domain by using the accelerated version of the "Discrete Fourier Transform" known as a "Fast Fourier Transform" or FFT. The `ifftf_fr32` function uses 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_fr32` may be used to initialize the array.

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 function may otherwise degrade due to data bank collisions.

The function uses static scaling of intermediate results to prevent overflow, and the final output therefore is scaled by `1/fft_size`.

Error Conditions

The `ifftf_fr32` function returns if the FFT size is less than eight or if the twiddle stride is less than one.

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 function uses 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);
```

ifftrad4

N-point radix-4 inverse input FFT

Synopsis

```
#include <filter.h>

void ifftrad4_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);
```

Description

This function transforms the frequency domain complex input signal sequence to the time domain by using the radix-4 Inverse Fast Fourier Transform.

The size of the input array `input`, the output array `output`, and the temporary working buffer `temp` is `fft_size`, where `fft_size` represents the number of points in the FFT. Memory bank collisions, which have an adverse effect on run-time performance, may be avoided by allocating all input and working buffers to different memory banks. 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 `3*fft_size/4` twiddle factors. The function `twidfftrad4_fr16` may be used to initialize the array. If the twiddle table contains more factors than needed for a particular call on `ifftrad4_fr16`, then the stride factor has to be set appropriately; otherwise it should be set to 1.

The arguments `block_exponent` and `scale_method` have been added for future expansion. These arguments are ignored by the function. To avoid overflow, the function performs static scaling by first dividing the input by `fft_size`.

This function is provided for backward compatibility with existing applications. New applications should use the `ifft_fr16` (on page 4-189) function instead.

Algorithm

The following equation is the basis of the algorithm.

$$x(n) = \sum_{k=0}^{N-1} X(k) W_N^{-nk}$$

Domain

Input sequence length `fft_size` must be a power of 4 and at least 16.

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 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 number of points in the FFT 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.

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_fr32.

Algorithm

The following equation is the basis of the algorithm.

$$c(i, j) = \frac{1}{n^2} \bullet \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} a(k, l) \bullet e^{-2\pi j(i \bullet k + j \bullet l)/n}$$

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          32
#define FFT_SIZE2           8
#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>

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_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)  \
   (state).c = (coeffs);   \
   (state).d = (delay);    \
   (state).k = (stages)
```

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 on page 4-209) 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.

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. See "#pragma align num" on page 1-280 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

$$D_m = X_m - A_2 * D_{m-2} - A_1 * D_{m-1}$$
$$Y_m = B_2 * D_{m-2} + B_1 * D_{m-1} + B_0 * D_m$$

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)
*/
section("L1_data_a") fract16 coeffs[NUM_STAGES * 5];
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 */
iir_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] * (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. 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 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)  \
   (state).c = (coeffs);  \
   (state).d = (delay);   \
   (state).p = (delay);   \
   (state).k = (2*(stages)+1)
```

The characteristics of the filter are dependent upon the filter coefficients and the number of stages. The A-coefficients and the B-coefficients for each stage are stored in a vector that is addressed by the pointer filter_state->c. This vector should be generated by the coeff_iirdf1_fr16 function (on page 4-120) for use with iirdf1_fr16, coeff_iirdf1_fx16 for use with iirdf1_fx16, coeff_iirdf1_fr32 for use with iirdf1_fr32, and by coeff_iirdf1_fx32 for use with iirdf1_fx32. The variable filter_state->k should be set to the expression (2*stages) + 1.

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` function.

Each filter should have its own delay line which is a vector of type `fract16` (for `iirdf1_fr16`), `_Fract` (for `irrdf1_fx16`). `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 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. For optimum performance, coefficient and state arrays should be allocated in separate memory blocks.

The `iirdf1` functions will adjust the output by the scaling factor that was applied to the `A`-coefficients and the `B`-coefficients by the `coeff_iirfd1` functions.

It is possible the filter's gain will cause the filtered response to be saturated. To avoid the saturation, the `B`-coefficients can be scaled *before* calling the `coeff_iirdf1` functions. For more information, refer to the example below.

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 = B0 * x(i) + B1 * x(i-1) + B2 * x(i-2)
y(i) = V + A1 * y(i-1) + A2 * y(i-2)
i = {0, 1, .., 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] = { . . . };

/* 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.

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`.

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.

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`.

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`.

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.

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 Table 4-11. Negative values for the phase argument are interpreted as a clockwise movement.

Table 4-11. Positive and Negative Phases for Fractional Polar Functions

Radians	Phase	
0	0.0	-1
$\pi/2$	0.25(0x2000)	-0.75
π	0.50(0x4000)	-0.5
$3/2\pi$	0.75(0x6000)	-0.25
$<2\pi$	0.999(0x7fff)	

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
`r` 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

N-point radix-2 real input FFT

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. 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 (on page 4-242) for use with rfft_fr16 or rfft_fx_fr16, and twidfftrad2_fr32 for use with rfft_fr32 or rfft_fx_fr32. For optimum 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 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 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 will 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 function scales the intermediate set of 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.

Any values for the argument `scale_method` other than 2 or 3 will result in the function performing static scaling.

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" on page 4-98 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_fr32(const 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);
```

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 will contain the following values:

- DC component of the signal in output[0].re (output[0].im = 0)
- First half of the complex spectrum in output[1] ...output[(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_fr32 may be used to initialize the array.

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 function 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.

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);

/* Add the 2nd half of the spectrum */
```

```
for (i = 1; i < (FFTSIZE/2); i++) {
  r_output[FFTSIZE - i] = conj_fr32(r_output[i]);
}
```

rfftrad4

N-point radix-4 real input FFT

Synopsis

```
#include <filter.h>

void rfftrad4_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);
```

Description

This function transforms the time domain real input signal sequence to the frequency domain by using the radix-4 Fast Fourier Transform. The `rfftrad4_fr16` function takes 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`, the output array `out`, and the temporary working buffer `temp` is `fft_size`, where `fft_size` represents the number of points in the FFT. To avoid potential data bank collisions, the input and temporary buffers should reside in different memory banks. This results in improved 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 `3*fft_size/4` twiddle factors. The function twidfftrad4_fr16 may be used to initialize the array. If the twiddle table contains more factors than needed for a particular call on rfftrad4_fr16, then the stride factor has to be set appropriately; otherwise it should be set to 1.

The arguments block_exponent and scale_method have been added for future expansion. These arguments are ignored by the function. To avoid overflow, the function performs static scaling by first dividing the input by fft_size.

This function is provided for backward compatibility with existing applications. New applications should use the rfft_fr16 (on page 4-225) function instead.

Algorithm

See "cfftrad4" on page 4-106 for more information.

Domain

Input sequence length fft_size must be a power of 4 and at least 8.

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 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 number of points in the FFT must be a power of 2 and must be at least 4 for `rfft2d_fr16` and at least 16 for `rfft2d_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, 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*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`, while `twidfft2d_fr32` may be used to initialize the arrays for `rfft2d_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.

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.

$$c(i, j) = \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} a(k, l) \bullet e^{(-2\pi \cdot (i \cdot k + j \cdot l))/n}$$

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 `rfft2d_fr16` and at least 16 for `rfft2d_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`.

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.

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 a radix-2 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 will contain 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.

Algorithm

This function takes FFT length `fft_size` as an input parameter and generates the lookup table of complex twiddle coefficients. The samples are:

$$twid\_re(k) = cos\left(\frac{2\pi}{n}k\right)$$

$$twid\_im(k) = -sin\left(\frac{2\pi}{n}k\right)$$

where:

n = `fft_size`
k = {0, 1, 2, ..., n/2 - 1}

Domain

The number of points in the FFT 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);
```

twidfftrad4

Generate FFT twiddle factors for radix-4 FFT

Synopsis

```
#include <filter.h>

void twidfftrad4_fr16(complex_fract16  twiddle_table[],
                      int              fft_size);

void twidfft_fr16(complex_fract16  twiddle_table[],
                  int              fft_size);
```

Description

The `twidfftrad4_fr16` function initializes a table with complex twiddle factors for a radix-4 FFT. The number of points in the FFT are defined by `fft_size`, and the coefficients are returned in the twiddle table `twiddle_table`.

The size of the twiddle table must be at least `3*fft_size/4`, the length of the FFT input sequence. A table can accommodate several FFTs of different sizes by allocating the table at maximum size, and then using the stride argument of the FFT function to specify the step size through the table.

If the stride is set to 1, the FFT function uses all the table; if your FFT has only a quarter of the number of points of the largest FFT, the stride should be 4.

For efficiency, the twiddle table is normally generated once during program initialization and is then supplied to the FFT routine as a separate argument.

The `twidfftrad4_fr16` function and the radix-4 FFT functions are only provided for backwards compatibility with existing applications. New applications should use one of the radix-2 FFT functions instead (see cfft, ifft, rfft). The twiddle table for the radix-2 FFT functions may be generated by calling `twidfftrad2_fr16`.

The `twidfft_fr16` function may be used as an alternative to the `twidfftrad4_fr16`. Both routines have the same functionality.

Algorithm

This function takes 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, ..., ¾n - 1}

Domain

The FFT length `fft_size` must be a power of 4 and at least 16.

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_fr16` function generates complex twiddle factors for the fast radix-4 FFT function `cfftf_fr16`, while the `twidfftf_fr32` function generates complex twiddle factors for the fast mixed-radix FFT functions `cfftf_fr32`, `ifftf_fr32`, `rfftf_fr32`, and `rfftf_fx_fr32`. 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.

A twiddle table of a given size will contain 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.

Error Conditions

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

Algorithm

The function calculates 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, ..., ¾n – 1}

Domain

The number of points in the FFT must be a power of 4 and must be at least 16 for `cfftf_fr16`, and a power of 2 and at least 16 for `cfftf_fr32`, `ifftf_fr32`, `rfftf_fr32` and `rfftf_fx_fr32`.

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);
```

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` may 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 may be used by any of the functions `cfft2d_fr32`, `ifft2d_fr32`, `rfft2d_fr32`, and `rfft2d_fx_fr32`.

A twiddle table of a given size will contain 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:

$$\frac{\textit{largest FFT size}}{\textit{current FFT size}}$$

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.

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`.

Error Conditions

The var_fr16 and var_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 var_fr32 and var_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.

Algorithm

The following equation is the basis of the algorithm.

$$c = \frac{n\sum_{i=0}^{n-1} a_i^2 - \left(\sum_{i=0}^{n-1} a_1\right)^2}{n(n-1)}$$

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.

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 ( )`

A PROGRAMMING DUAL-CORE BLACKFIN PROCESSORS

The Blackfin processor family includes dual-core processors, such as the ADSP-BF561 processor. 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 VisualDSP++.

The appendix begins with a brief comparison of the single-core versus dual-core Blackfin processors, before describing VisualDSP++ recommended approaches to application development. Finally, it offers guidelines for developing systems on dual-core Blackfin processors. The appendix expects users to have an understanding of programming for multiple processors/threads.

All examples given are for the ADSP-BF561 processor.

The appendix contains:

- "Restrictions on Dual-Core Applications" on page A-25
- "Dual-Core Programming Examples" on page A-26
- "Synchronization Functions" on page A-43

For the most efficient use of information in this appendix, you should be familiar with the Blackfin architecture and have experience in building and executing C/C++ applications for the Blackfin architecture within the VisualDSP++ environment. The appendix focuses only on the additional considerations necessary for dual-core programming.

Dual-Core Blackfin Architecture Overview

Each dual-core Blackfin processor has two Blackfin cores (core A and core B), 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 so on. On reset, core A starts running from its reset address, while core B is disabled. Core B starts running when it is enabled by core A.

VisualDSP++ enables core B when it connects to an EZ-KIT Lite board, as part of the program download process.

When core B starts running, it runs its own application from its own reset address.

The two cores may 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 ADSP-BF561 hardware reference for detailed information on the ADSP-BF561 processor's architecture.

Approaches Supported in VisualDSP++

VisualDSP++ supports three different approaches to project development for dual-core Blackfin processors:

- *Single-core applications*
  In this approach, only core A is used, and core B remains disabled. (See "Single-Core Application" on page A-5.)
- *One application per core*
  In this approach, each core is treated as a separate processor, built individually. The VisualDSP++ project explicitly builds a `.dxe` file for a particular core. Resource sharing is coarse-grain and is managed by the developer. (See "One Application Per Core" on page A-7.)
- *One application across both cores*
  In this approach, a hierarchy of VisualDSP++ projects builds a single application that supports both cores. Resource sharing is fine-grain, managed by the linker. (See "Single Application/Dual Core" on page A-16.)

The following sections describe these approaches in more detail.

The approaches represent increasing levels of sophistication, with corresponding levels of complexity.

A single-core application allows the processor to be used as a migration path from other Blackfin processors and as a means of running standard and legacy applications with minimal effort. Benchmarks are typical examples. This simplistic approach does not exploit the full potential of the dual-core Blackfin processor but provides the fastest route for getting existing code "up and running."

Having one application per core extends this simplistic approach to use both cores. Effectively, two single-core applications are built independently and run in parallel on the processor. The shared memory areas, both internal and external, are each sub-divided into three areas—a section dedicated to core A, a section dedicated to core B, and a shared section. It is left up to the developer to arrange for shared, serialized access to the shared areas from each of the cores.

The single-application/dual-core approach is the most powerful, because it allows for all of the shared memory areas to be used efficiently by both cores. Common code can be placed in shared memory to avoid duplication. Shared data can be placed in shared memory without the need for explicit positioning. This approach allows an expert developer to exercise fine control over the structure of the application, using the VisualDSP++ advanced linker capabilities.

The VisualDSP++ libraries and `.ldf` files provide support for multi-core builds, used by the latter two approaches. This support is available through the `-multicore` compiler switch and through the `__ADI_MULTICORE` linker macro. (See "-multicore" on page 1-50.)

The VisualDSP++ **Project Wizard** provides support for generating `.ldf` files, startup code and template files for your project, for each of the supported approaches. The resulting files are customized according to your project options.

Single-Core Application

The single-core application approach is supported by the default compiler linker description file (`.ldf`). Whenever the compiler is asked to generate an executable file without specifying an `.ldf` file, the compiler uses a default `.ldf` file for the platform in question. For example,

```
ccblkfn -proc ADSP-BF561 prog.c -o prog.dxe
```

does not specify an `.ldf` file, so the compiler uses the default, whereas:

```
ccblkfn -proc ADSP-BF561 prog.c -o prog.dxe -T ./my.ldf
```

directs the compiler to use `./my.ldf` as the `.ldf` file.

The default compiler `.ldf` file for the ADSP-BF561 processor is located in `<install_path>/Blackfin/ldf/`. It is similar to the corresponding default `.ldf` files for other Blackfin processors, such as the ADSP-BF533 processor (although with a different memory map).

The `.ldf` file creates a single `.dxe` file that runs on core A. By default, the same `.ldf` file is also used for the one-application-per-core build, described in "One Application Per Core" on page A-7.

You can create a project of this kind using the Project Wizard, if you select **File, New, Project....** **Dual-core Settings** offers two choices for single-core projects. In **Single core: Single application**, only core A runs, while core B remains disabled. In **Dual core: Single application**, a default program runs on core B, placing it into `IDLE` mode, so that core A can change processor speed.

When you add or customize an `.ldf` file via the **Project Options** dialog box, a single-core application `.ldf` file is produced if you select **Core A** under **LDF Settings**, **Multi-Core Selection**.

There is an example of this approach on page A-26.

Shared Memory

The `.ldf` file divides the 128 Kbytes shared L2 internal memory as follows:

- Lowest 32 Kbytes: reserved for core B, not used in this approach
- Next 32 Kbytes: reserved for core A, usable via section `l2_sram_a`
- Most of remaining 64 Kbytes: reserved for shared data, usable via section `l2_shared`
- 16 bytes reserved for synchronization locks, not used by this approach
- 1 Kbytes reserved for second-stage boot loader, not used by `<install_path>\Blackfin\LDF`

Note that the lowest 64 Kbytes are partitioned between the two cores. This is because the same `.ldf` file is also used for the "one application per core" approach described later. For a single-core application, it may be desirable to customize the `.ldf` file so that all of L2 internal memory is available for core A, although this will complicate migration towards a multi-core solution.

To place code or data into the area reserved for core A, place them into the `l2_sram_a` section.

External memory is shared between the cores and can be used via the section `sdram_data`.

Synchronization

Synchronization is not necessary for the single-core approach. The `.ldf` file still reserves a section of internal L2 memory for synchronization locks, for backward compatibility.

Cache, Startup, and Events

For a single-core application, normal cache configuration and event handling is used, such as for the ADSP-BF533 processor. The only difference is that the `.ldf` file maps a cacheability protection lookaside buffers (CPLBs) configuration table explicitly for each core. Where `ADSP-BF533.ldf` links against `cplbtab533.doj`, a single-core ADSP-BF561 processor application would link against `cplbtab561a.doj`, for core A's CPLB configuration.

The run-time header executed on startup is a generic routine that has been assembled for the ADSP-BF561 processor. It behaves in the same manner as for other Blackfin platforms, except that it makes no attempt to modify the clock speed. It enables cache, interrupts and exceptions in the same fashion as for other Blackfin processors.

Creating Customized .ldf Files

To create a customized `.ldf` file for a single-core application, under **Project Options**, select **Add Startup code/LDF**. Under **LDF Settings**, **Multi-core Selection**, choose **Core A**.

One Application Per Core

Like the single-core application approach, the one-application-per-core approach can use either customized `.ldf` files or the default compiler `.ldf` file. In this chapter, it is called *per-core*.

There is an example of this approach on page A-27.

Using the Default Compiler .ldf File

The default compiler `.ldf` file builds one application for each invocation, either for core A or for core B, according to command-line options. To produce the two applications, first build the application for one core, and then build the application for the second core.

For example,

```
ccblkfn -proc ADSP-BF561 -flags-link -MDCOREA -o \
        p0.dxe a1.c a2.c

ccblkfn -proc ADSP-BF561 -flags-link -MDCOREB -o \
        p1.dxe b1.c b2.c
```

This would build two applications— `p0.dxe` for core A and `p1.dxe` for core B.

The `COREA` and `COREB` linker flags define preprocessor macros that select alternative `PROCESSOR` directives in the `.ldf` file. If neither `COREA` nor `COREB` is defined, the `.ldf` file automatically defines `COREA` and links for core A. This is how the single-core application (described in "Single-Core Application" on page A-5) is implemented.

ⓘ If you create the projects using the Project Wizard, the `COREA` and `COREB` preprocessor macros will be added for you by the Project Wizard.

Using Customized .ldf Files

When using customized `.ldf` files, you create a customized `.ldf` file configured for each core. Create a project for each application you are building (one for core A, one for core B).

ⓘ You can create these projects using the Project Wizard, if you select **File, New, Project... Under Dual-core Settings**, then select **Dual core: one application per core**.

Once you have created your projects, add a customized `.ldf` file to each:

1. For each project, go to **Project Options**, **Add Startup code/LDF**.
2. Under **LDF Settings**, **Multi-core Selection**, choose either **Core A** or **Core B**, as appropriate for the project.

VisualDSP++ creates customized `.ldf` files for each core, containing only the parts relevant to the core in question. Consequently, you do not need to specify `COREA` or `COREB` when linking the applications.

Shared Memory

The memory map for the default `.ldf` file defines all of the internal memories for both cores, although the `PROCESSOR` section uses only the areas defined for the currently-selected core. Thus, while `COREB` is defined, the `.ldf` file maps the `L2_sram_a` section into the lowest 32 Kbytes of the L2 internal memory. When `COREA` is defined, it maps the `L2_sram_b` section into the next 32 Kbytes of the L2 internal memory. In this manner, the two separate builds can map code or data into the common L2 internal memory without conflict.

Customized `.ldf` files define only the areas of memory for the core in question; therefore, a customized `.ldf` file for core A maps section `L2_sram_a`, while a customized `.ldf` file for core B maps section `L2_sram_b`.

Sharing Data

The .ldf files provide two shared data areas, l2_shared and sdram_shared, which are in L2 memory and SDRAM respectively. For the per-core approach, the recommended method for sharing data is as follows:

For core A's project:

- Define the data items to be shared in a source module that contains only shared items (that is, no items to be mapped to core-specific memory).
- Declare the data items to be volatile.
- Set the file attribute sharing to MustShare for the shared-data module.
- In the source module, declare the shared-data items to be part of a section that is shared by both cores, such as l2_shared or sdram_shared.

For example,

```
#include <ccblkfn.h>
#pragma file_attr("sharing=MustShare")
section("l2_shared") volatile char shared_buffer[1024];
section("sdram_shared") volatile testset_t lock_variable;
```

For core B's project:

- Declare the data items as external (via `extern`), as they will be supplied by the definitions from core A's project.

```
extern volatile char shared_buffer[];
extern volatile testset_t lock_variable;
```

- In **Project Options**, **Link**, **LDF Preprocessing**, **Preprocessor Macro Definitions**, define the macro `OTHERCORE`. (You do not need to supply a value.) If you create your project using the Project Wizard, this macro will be added to your project automatically on creation.

- Add a header file, `local_shared_symbols.h`. This file should redefine the `OTHERCORE` macro to be the pathname to the `.dxe` file produced by core A's project, and include the library header file `shared_symbols.h`. For example,

```
#undef OTHERCORE
#define OTHERCORE "Release/Core A.dxe"
#include <shared_symbols.h>
```

- For each data item to be shared, add a `RESOLVE()` command to `local_shared_symbols.h`, giving the symbol name and the `OTHERCORE` macro, for example:

```
RESOLVE(_shared_buffer, OTHERCORE)
RESOLVE(_lock_variable, OTHERCORE)
```

> The `RESOLVE` commands will be processed by the linker, and therefore they must use the linkage name of the symbols. For C declarations, this typically means prefixing the symbol name with an underscore. C++ symbol names are "mangled" by default to encode the additional type information. If you are sharing C++ objects, you can declare them using `extern "C"` to give them C linkage instead.

The default and generated `.ldf` files for core B recognize the `OTHERCORE` macro, and include the `local_shared_symbols.h` header file into the `.ldf` file when it is defined. When building core B's `.dxe` file, the linker will not have a local definition for the shared data items. This is because they have been mapped only when building core A's `.dxe` file, and not when building core B's. Therefore, the linker follows the `RESOLVE` directives in the included file to resolve the specified symbols to the same address as used for core A.

The `shared_symbols.h` header file, included by `local_shared_symbols.h`, is a VisualDSP++ header file that gives suitable `RESOLVE` directives for the run-time library's shared symbols. It uses the macro `OTHERCORE` to identify the `.dxe` file to be used. For more information, see "One Application per Core Example" on page A-27.

Data shared between the two applications must be declared as `volatile`, so that the compiler does not cache values in registers during times when the other core might be updating the value.

The data caches within cores A and B do not maintain coherence, so two alternatives are available:

- Do not enable data caching for shared areas.
- After finishing an access, but before releasing the data to be used by the other core, flush the data from the cache and invalidate the corresponding cache entries.

Sharing Code

To share code between applications, follow the same steps as for sharing data (on page A-10):

1. Map the functions to the shared area in the application for core A.
2. In the application for core B, declare the functions as external (via `extern`).
3. Add `RESOLVE` directives to the `local_shared_symbols.h` header file for core B, giving the functions' external names.

Shared Code With Private Data

It is sometimes desirable for a function to maintain its own private data. In a single-threaded, single-core application, declare the data as `static`.

In a dual-core application where the code is shared, the same data is used by both cores. If you want each core to have its own instance of the private data, use library routines provided with VisualDSP++ to allocate private copies of the data. These routines are described in detail in "adi_obtain_mc_slot, adi_free_mc_slot, adi_set_mc_value, adi_get_mc_value" on page 3-76.

Synchronization

Synchronization functions exist in the run-time library for claiming and releasing a lock variable. They are described in detail in "adi_acquire_lock, adi_try_lock, adi_release_lock" on page 3-71.

```
#include <ccblkfn.h>
void adi_acquire_lock(testset_t *t);
int adi_try_lock(testset_t *t);
void adi_release_lock(testset_t *t);
int adi_core_id(void);
```

Cache, Startup, and Events with Default .ldf Files

Each core has its own caches and its own cache configuration table. These are linked in by the `.ldf` file according to whether `COREA` or `COREB` is defined. `COREA` links against `cplbtab561a.doj`, while `COREB` links against `cplbtab561b.doj`.

Each application has its own copy of the `___cplb_ctrl` cache configuration variable. Each application also has its own definitions of the guard symbols that the `.ldf` file defines to indicate whether L1 SRAM spaces are available for cache use. Thus, the two applications can run with entirely independent cache configurations. The section `cplb_code` must be mapped into L1 Instruction memory so that the CPLB configuration routines can access these core-specific guard symbols.

However, the startup code is the same for the two cores. In other words, each application in the per-core approach receives its own copy of the same startup code, resolved to the `Reset` address of that core. In particular, the default startup code does not include any functionality to allow core A to enable core B. Use the following function to enable core B:

```
#include <ccblkfn.h>
void adi_core_b_enable(void);
```

> VisualDSP++ also arranges for core B to be enabled when downloading applications to the EZ-KIT Lite boards.

Each core registers its own event handler (for CPLB events, if requested), and handles interrupts and exceptions separately. The two applications can have separate event masks. Signals can be passed between the two applications by triggering interrupts via the system interrupt controller. The run-time library allows interrupt handlers to be registered, but does not provide direct support for the system interrupt controller, or for raising events at that level.

Cache, Startup, and Events with Customized .ldf Files

The cache configuration for both applications is managed by **Project Options**, **Startup Code Settings**, **Cache and Memory Protection**. Using **Project Options** ensures that the start-up code invokes only the CPLB configuration routines where necessary, and that the L1 memory usage matches the cache options selected.

The start-up code is essentially the same for the two cores, but each application receives its own generated start-up routine according to **Project Options**, so there may be some differences. Note that the default startup code does not include functionality to allow core A to enable core B. You should arrange for core A to do this when your application is suitably configured.

A convenient way to enable core B is to use the following function:

```
#include <ccblkfn.h>
void adi_core_b_enable(void);
```

> VisualDSP++ also arranges for core B to be enabled when downloading applications to the EZ-KIT Lite boards.

Each core registers its own event handler (for CPLB events, if requested), and handles interrupts and exceptions separately. The two applications can have separate event masks. Signals can be passed between the two applications by triggering interrupts via the system interrupt controller. The run-time library allows interrupt handlers to be registered, but does not provide any direct support for the system interrupt controller or for raising events at that level.

Single Application/Dual Core

The single application/dual core approach generates a single application with just one build process. The application is divided into three components: the two individual cores and the shared memory. (For the purposes of the build, all common memory is treated as one.)

The single application/dual core approach (also known as single/dual) allows a more complex application to be built. This is because the three major components are produced during a single linking process that resolves all symbols at once. This process allows code and data in the shared memories to be referenced directly from the cores, allowing the cores to use the same instance of a function or data item.

This sharing process makes use of more advanced linker facilities that are not normally required or employed for single applications that run on a single core. These extra capabilities can present a steep learning curve for those new to cross-system linking. Therefore, the single/dual approach adopts a set of conventions to assist in the development of dual-core applications. The `.ldf` files generated by VisualDSP++ rely on these conventions for simplicity. The advanced developer may choose alternative approaches by using entirely customized `.ldf` files.

There is an example of this approach on page A-30.

Target Conventions

The conventions are as follows.

- The application is arranged as a hierarchy of targets, as shown by Figure A-1, with the final application being the top-level project. This top-level target is of type “DSP executable”.
- Beneath the top-level target, there are four sub-targets: core A, core B, shared internal L2 memory, and shared external memory areas. These sub-targets are of type “DSP library”.

- The sub-targets create individual files called `corea.dlb`, `coreb.dlb`, `sml2.dlb` and `sml3.dlb`.
- The top-level target links against the libraries generated by the sub-targets, resolving symbols across all of the system at once, and produces three output files: `p0.dxe`, `p1.dxe` and `L2_and_L3_common_memory.sm`. These files may be loaded into the Blackfin processor.

Figure A-1 shows a typical five-project setup.

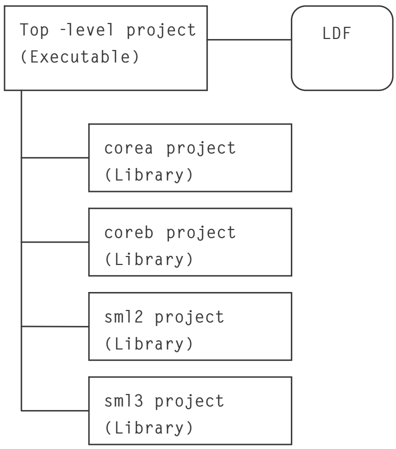


Figure A-1. Five-Project Setup

With the application divided into individual libraries, it is simpler to arrange for a part of the application to reside within a particular core or within a particular shared memory.

Establishing a convention for file names (`p0.dxe`, `sml2.dlb`, and so on) means that the `.ldf` file in the top-level target can use the output of a sub-target without needing customization.

Using file attributes is an alternative approach. (For more information, see "File Attributes" on page 1-471.) This approach allows you to control memory placement without needing several sub-projects. This approach is described on page A-22, with an example shown on page A-37.

You can create these projects using the Project Wizard, if you select **File, New, Project... Under Dual-core Settings**, select **Dual core: Single application using both cores**. The Project Wizard will also create a customized `.ldf` file and startup code.

Multi-Core Linking

The single/dual approach uses advanced linker facilities to resolve cross-references between the cores and shared memories. Each core is described by a `PROCESSOR` directive, and the two shared memory areas (the internal L2 memory and the external memory) are described by a single `COMMON_MEMORY` directive. The `COMMON_MEMORY` region uses the `MASTERS` directive to indicate that the two `PROCESSOR` directives are attempting to resolve external references through the `COMMON_MEMORY` region.

Both the `PROCESSOR` directives and the `COMMON_MEMORY` region can link against libraries, as shown by Figure A-2. The `PLibs` libraries are mapped directly by the `PROCESSOR` directives. If an external reference is resolved using these libraries, the definition is mapped into the private memory of core A or core B, as appropriate. The libraries shown as `CLibs`, are mapped by the `COMMON_MEMORY` region. If an external reference is resolved using these libraries, the definition will be mapped into the `COMMON_MEMORY` region, and may be shared between core A and core B.

For information on these linker facilities, refer to the *VisualDSP++ 5.0 Linker and Utilities Manual*.

When linking a single-application/dual-core application, you must use the `-multicore` switch to ensure that the run-time libraries use the correct synchronization locks.

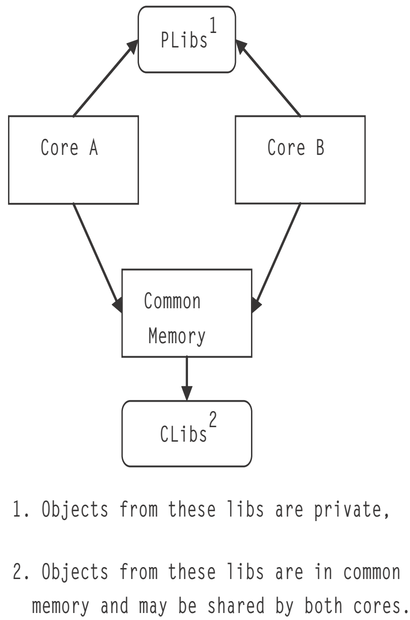


Figure A-2. Dual-Core Linking

Creating the .ldf File

The single/dual approach requires a custom `.ldf` file. This is because the default `.ldf` files for the dual-core Blackfin processors are designed for the simpler single-core and per-core approaches. It is not necessary to modify the `.ldf` file in any way, once created. The sub-projects do not require `.ldf` files.

The easiest way to create the custom `.ldf` file is to use the Project Wizard: select **File, New, Project... Under Dual-core Settings**, select **Dual core: Single application using both cores**. The custom `.ldf` file will be created, along with the project hierarchy.

Alternatively, you can create the custom `.ldf` file via **Project Options** for the top-level project, using **Add Startup code/LDF**. Ensure that the **Cores A and B** option is selected under **LDF Settings, Multi-core Selection**.

Shared Memory

Code and data can be mapped into internal L2 memory by placing them into the `sml2` sub-target. The `.ldf` file links the `COMMON_MEMORY` area against the library produced by this sub-target. The usual sections (`program`, `data1`, `constdata`, and so on) are mapped, as is `l2_sram`.

Code and data can be mapped into the external memory by placing them into the `sml3` sub-target.

Shared Data

To share data items between the two cores, do the following:

- Define the shared-data items in a source module that contains only shared items (that is, do not include any code or data that will be private to one of the cores).
- Make the source module part of the `sml2` or `sml3` sub-project, as appropriate.
- Within the source module, define the file attribute `sharing`, with the value `MustShare`, that is,
  `#pragma file_attr("sharing=MustShare")`
- Declare the data items to be `volatile`.

Sharing Code

Application code may be shared between the two cores by following the same steps as for sharing data (on page A-20).

If run-time library functions are to be shared, then the libraries in which they reside must only be included in the `CLibs` list of libraries (as shown in Figure A-2 on page A-19). In other words, they should not be in the list of libraries linked-against by the `PROCESSOR` directives. Otherwise, the cores will link in their own copy of the function instead of using the shared version in `COMMON_MEMORY`.

Synchronization

Synchronization between the cores can be achieved as for the per-core approach using "adi_acquire_lock, adi_try_lock, adi_release_lock" on page 3-71. The synchronization lock variables must be defined in the `sml2.dlb` or `sml3.dlb` sub-targets, so that they are mapped into the shared memory.

Cache, Startup, and Events

The generated `.ldf` file for the single/dual approach maps a copy of the startup code into each core, resolving the copies to the `Reset` addresses of the cores. Startup, cache configuration, and events are as for the per-core approach.

Only a single startup code routine is generated and built, and linked into both cores. Ensure that your Project Options are suitable for both cores.

The generated `.ldf` file also maps the `cplb_code` section into the L1 instruction memories of the cores. This means that the definitions of the guard symbols are local to the processor. If the `.ldf` file is changed so that the `cplb_code` section is mapped into shared memory instead, then the `COMMON_MEMORY` directive must also define appropriate guard symbols, otherwise the link may resolve the reference by importing the default guard symbols from the run-time library.

Dual-Core Applications That Use File Attributes

The five-project convention provides a basic organizing tool for managing code and data placement within a dual-core system.

Using file attributes is an alternative approach. (For more information, see "File Attributes" on page 1-471.) This approach allows you to control memory placement without needing several sub-projects. An example is shown in "Interprocedural Analysis and File Attributes" on page A-37.

The generated dual-core `.ldf` files support the following file attributes by default:

- `DualCoreMem`: May have the values `CoreA` or `CoreB`. This attribute is used to filter the command-line objects so that items for one core do not get mapped to the other.
- `prefersMem`: May have values `internal` or `external`, in which case the linker will attempt to map the objects accordingly. Other values are equivalent to not setting this attribute.
- `sharing`: Objects with this attribute set to `MustShare` will be subjected to additional checking by the linker, to ensure that there is only a single definition of the symbols defined by the object.

In addition, the run-time library defines a number of file attributes for each supported function. (See "Library Attributes" on page 3-8 for more information.) These can all be used when mapping library routines to dual-core systems to help with code and data placement.

File attributes allow you to link dual-core applications without organizing core-based objects into libraries. This allows you to make more use of interprocedural analysis (which has reduced benefit with library objects).

To use attributes for dual-core linking, do the following:

1. Distribute your sources between the two cores by defining attributes `DualCoreMem=CoreA` or `DualCoreMem=CoreB` as required. These sources can be part of your top-level project – they do not need to be in `corea` or `coreb` sub-projects.

2. Objects that will be mapped into common memory must be built into a library (because only libraries can be mapped into `COMMON_MEMORY`). This can be via the `sml2` or `sml3` sub-projects, or via another project.

3. To avoid link-time errors, create an empty C file and add it to each of the standard sub-projects you are not using. This will allow VisualDSP++ to create the expected libraries that will be referenced at link-time, thus avoiding to have to manually modify the `.ldf` file.

For more information, see “Interprocedural Analysis and File Attributes” on page A-37.

Run-Time Library Functions

The three approaches discussed here are concerned primarily with arrangement of application code and data, but it is a rare application that does not make use of run-time library support in some manner. This raises complications for a dual-core system.

Re-Entrancy

The majority of run-time library routines make no use of private data, operating on parameters and stack data only. Such functions are fully usable within a dual-core system without the need for locking. Some Standard routines—such as `strtok()`— use private data, and some routines update global data—the `errno` variable being the most common global variable so effected.

Multi-core applications must be built with the `-multicore` compiler switch, which means that the multi-core variants of these functions will be used. They have the appropriate locking enabled, and allocate per-core private copies of such data to ensure that each core sees standardized behavior.

> The `-multicore` switch has two settings under **Project Options**. The **Will be linked with re-entrant libraries** option under **Compile, Processor (2)** sets the `-multicore` switch at compile-time. The **Use re-entrant multicore libraries** option under **Link, Processor** sets the `-multicore` switch at link-time. These flags are set automatically if you create your project(s) using the Project Wizard.

However, not all run-time library functions may be freely mapped. There are some restrictions on mapping. These are documented in "Library Function Re-Entrancy and Multi-Threaded Environments" on page 3-14.

Placement

Use the run-time libraries' file attributes to control placement of library components among core A, core B and common memory. This approach is more effective than using the normal section-based placement, as the majority of library components are mapped into the standard sections.

For more details on the run-time libraries' attribute support, see "Library Attributes" on page 3-8.

For restrictions on placing library functions in memory, see sections "Library Placement" on page 3-18 and "Section Placement" on page 3-19.

Restrictions on Dual-Core Applications

There are some restrictions for dual-core applications that do not apply to other applications.

Compiler Facilities

The following features have some restrictions with dual-core systems:

- Interprocedural analysis (IPA) optimization requires you to use `#pragma core` (on page 1-304) to identify distinct symbols that are defined differently for each core. For more information, see "Interprocedural Analysis and File Attributes" on page A-37.
- Profile-guided optimization (PGO) requires you to use session IDs to distinguish between profiles gathered for each core. For more information, see "Profile-Guided Optimization in Dual-Core Systems" on page A-32.
- Instrumented-code profiling (`-p[1|2]` compiler switches on page 1-65) is not supported.

Cross-Core Memory References

It is not valid for code executing in one core to access the L1 memory of the other core, whether for code or data references. Attempts to do so will raise an exception. Therefore, when pointers to L1 memory are stored in shared memory and accessed by common code, care must be taken to ensure that such pointers are not de-referenced by the other core. This applies to both the per-core and single/dual approaches.

The linker's `COMMON_MEMORY` construct provides some protection against this situation. In most cases the linker can resolve such cases without the need for user interaction, by duplicating input sections. Where the linker cannot safely resolve the situation, a link-time error occurs.

See the *VisualDSP++ 5.0 Linker and Utilities Manual* for more information.

Dual-Core Programming Examples

The following examples show the different code design approaches as applied to a simple client-server application on the ADSP-BF561 processor. The client passes a list of sentences to a server, one by one, and the server encodes them via a trivial `ROT13` algorithm. The client shows each string before encoding, after encoding, and once more after re-encoding (which, under `ROT13`, restores the original plain text).

A `frame` object is used to pass each sentence between the client and the server, and to return the encoded form.

The examples can be found in the VisualDSP++ installation directory, under:

```
Blackfin/Examples/No Hardware Required/Compiler Features
```

Single-Core Application Example

The single-core approach can be found in the `Rot13 Single-Core` project in the `Rot13 Single-Core` directory. It is a single-threaded version, for simplicity. Since there is just a single thread, no synchronization is necessary. The `main()` function for core A is in the file `maina.c` and calls the `rot13()` function directly. This example does not enable core B and serves as a comparison with multi-core variants.

The example was created with the Project Wizard, using **Single core: Single application**. It makes use of the default `.ldf` file, and since it does not define any preprocessor macros, links for core A by default.

One Application per Core Example

The per-core approach is in the project group `Rot13 Per-Core` in the `Rot13 Per-Core` directory. Since it requires synchronization, locking routines are added to the build. It also requires another thread that runs in the second core. The two threads use a lock to serialize access to the buffer and a protocol to indicate the buffer state. The procedure is as follows:

1. The buffer starts in state `ProcessingDone`. There is no work pending.
2. Core A copies data into the buffer and sets the state to `WaitingToBeProcessed`.
3. The buffer belongs to core B, which does the necessary encoding and resets the state to `ProcessingDone`.
4. The buffer now belongs to core A again, and core A is free to examine the results.
5. When core A has passed all packets of data to core B and received all the responses, core A sets the state to `NoMoreWork`. This indicates to core B that it can terminate.

There are two projects: a client (`Rot13 Per-Core_CoreA`) and a server (`Rot13 Per-Core_CoreB`). The projects were created by the Project Wizard, using **Dual core: One application per core**.

The client consists of `client.c` and `report.c`, which contain core A's `main()` function and the display routine `report()`.

The server consists of `server.c` and `rot13.c`, which contain core B's `main()` function and the encoding/decoding `rot13()` function.

Core A's project also contains the source files for the shared data (the frame and a communications lock) and the locking routines. The shared data is declared as volatile, to prevent the optimizer from making assumptions about values. The client .ldf file maps objects from these shared source files into regions of memory accessible by both cores.

The server project does not contain these shared sources. Instead, it declares the shared data and functions as external. Since the project contains no definitions for the shared elements, the linker has to resort to outside sources to resolve the symbols during linking. The server project's .ldf file includes the file local_shared_symbols.h, which contains the following code:

```
#undef OTHERCORE
#ifdef DEBUG
#define OTHERCORE "Debug/Rot13 Per-Core_CoreA.dxe"
#endif
#ifdef RELEASE
#define OTHERCORE "Release/Rot13 Per-Core_CoreA.dxe"
#endif
#include <shared_symbols.h>
RESOLVE(_corelock, OTHERCORE)
RESOLVE(_frame, OTHERCORE)
RESOLVE(_claim_lock, OTHERCORE)
RESOLVE(_release_lock, OTHERCORE)
```

These contents instruct the linker that it should resolve the external references by examining the file OTHERCORE – the .dxe produced by the client project – and resolving the symbols to the same addresses as used in that other executable. This means that the source components common to both projects are resolved to the same address in both executables. The DEBUG and RELEASE macros are set by the configurations generated by the Project Wizard, so that the linker resolves against the executable in the appropriate directory.

The shared_symbols.h file also resolves common symbols in this manner. It lists symbols from the run-time library that must be common to both cores in a multi-core application.

Each project has a custom `.ldf` file, generated automatically by the Project Wizard. Note the following points:

- As these `.ldf` files are generated for a dual-core processor, the multi-core settings have to be selected accordingly. The client's `.ldf` file is for core A, while the server's is for core B.
- Both projects are flagged as being linked with re-entrant libraries, under:
  - **Compile, Processor**(2). This setting affects header-file preprocessing during compilation.
  - **Link, Processor**. This setting affects the library selection during linking.
- External memory is enabled under the following:
  - **LDF Settings, External Memory**.
  - **Link, Processor**.
- The server project has the client project as a dependency, so the client project will automatically be built if required when building the server project.

To build and use the example project, do the following:

1. Create a session in the VisualDSP++ IDDE for the ADSP-BF561 Blackfin processor.
2. Open the `Rot13 Per-Core.dpg` project group.
3. Make `Rot13 Per-Core_CoreB` the active project.

4. Rebuild all.
5. Ensure that when loading the resulting executables:
   - `Rot13 Per-Core_CoreA.dxe` is loaded into core P0
   - `Rot13 Per-Core_CoreB.dxe` is loaded into core P1

Single Application/Dual-Core Example

The sources for the single/dual approach are effectively the same as for the per-core approach. The differences appear in how they are linked into a single application. The example is in the `Rot13 Dual-Core` project group, in the `Rot13 Dual-Core` directory.

Five projects are used, created by the Project Wizard: the overall project (`Rot13 Dual-Core`) and four sub-projects (`corea`, `coreb`, `sml2` and `sml3`). These sub-projects are all dependencies of the main `Rot13 Dual-Core` project.

The main project has an `.ldf` file, generated through the Project Wizard. Note that:

- Under **LDF Settings, Multi-core Selection** is set to **Cores A and B**.
- External memory is enabled under:
  - **LDF Settings, External Memory**
  - **Link, Processor**
- Re-entrant libraries are selected under:
  - **Compile, Processor**(2)
  - **Link, Processor**

The source files are distributed among the sub-projects in the following manner:

```
Rot13 Dual-Core: No sources
corea: client.c report.c
coreb: server.c rot13.c
sml2: lockfns.c lockdata.c
sml3: frame.c
```

This division between `sml2` and `sml3` is arbitrary and is used to demonstrate placement within the different shared memories. The `lockdata.c` and `frame.c` files contain the shared symbols, while `lockfns.c` contains the shared code.

The entire application is built using the following single build process:

1. Create a session in the VisualDSP++ IDDE for the ADSP-BF561 Blackfin processor.
2. Open the `Rot13 Dual-Core.dpg` project group.
3. Make `Rot13 Dual-Core` the active project.
4. Rebuild all.
5. Ensure that, when loading the resulting executables:
   - `P0.dxe` is loaded into core P0.
   - `P1.dxe` is loaded into core P1.

First, the sub-target libraries are built, then the top-level target is used to build the whole application. The `.ldf` file specifies all the output files within it, generating `p0.dxe`, `p1.dxe` and `L2_and_L3_common_memory.sm`.

Profile-Guided Optimization in Dual-Core Systems

For single-core applications on a dual-core system, profile-guided optimization (PGO) is used in the same way as it is for any other single-core system. Since the second core is not being used, it has no effect on PGO usage.

When you are using a dual-core system, whether via the per-core approach or via the single/dual approach, PGO usage is different because the VisualDSP++ IDDE graphical interface to PGO is designed for a single-core system. The IDDE understands that, for PGO, the application must be:

- Built using `-pguide` (on page 1-67) to prepare for profile-gathering
- Executed in the simulator using input data sets, to gather the profile
- Rebuilt using the resulting profile, to obtain the best optimization

To this end, the IDDE automates the process of building, executing and rebuilding, but does so in a manner that assumes all input data sets are being fed to a single executable. On a dual-core system, there are two executables, one per core, and the distribution of input data between them is not predictable. Therefore, the IDDE's automated PGO interface is not suitable.

Command-Line Profile-Guided Optimization

To run PGO on a dual-core system, use the `pgoctrl` command-line tool. This tool enables and disables profile gathering. You will have to arrange for each executable to read its input data sets as necessary. Use the `pgoctrl` tool as follows:

```
pgoctrl on path-to-profile-file.pgo
pgoctrl off
```

These commands must be entered while the applications are already loaded into a simulator session within the IDDE. There must only be one instance of VisualDSP++ active during this time.

The first command enables profile-gathering and informs the IDDE of the file name into which the profile-data will be stored. From this point on, whenever the program executes within the simulator, PGO will be counting the times the program passes through paths of control.

Having enabled PGO, you can run your application for the required time.

The second command terminates profile-gathering. The IDDE writes the gathered profile to the named file and stops counting path execution.

> If the file already exists, it will not be overwritten. Instead, the existing file's contents will be merged with the new profile data. To create entirely new profiles, ensure that the file name specifies a new file rather than an existing one.

> Profiling is not a persistent state. If you terminate the VisualDSP++ session and later restart VisualDSP++, you will need to re-enable profiling, if it is still required.

PGO Session Identifiers

In a dual-core system, sometimes the same source module is used in both cores. The source module may be compiled with different options, or it may be compiled once to an object file and then linked into the private core areas of memory. In such cases, the module should ideally be profiled separately for each core, and then re-optimized differently according to each core's execution profile. This is achieved using PGO session identifiers (*session IDs*).

Session IDs are used to distinguish between two or more counters for the same source-level symbol in an application. For example, both cores will have a `main()` function and those functions are likely to be different. Each `main()` is assigned a different session ID during initial compilation and these IDs are recorded in the gathered profiles. Then, when recompiling, the session IDs are used to associate the gathered counts with the particular version of `main()` being recompiled.

You specify session IDs using the `-pgo-session` switch (on page 1-67), during both initial compilation and during recompilation. Each use of the source module in the application must have a different session ID. This means that, rather than compiling once and then linking into both cores, you must recompile for each instance linked into the application (even if the only difference is in the session ID).

Example of Dual-Core Profile-Guided Optimization

"Example of Profile-Guided Optimization" on page 2-37 demonstrates how PGO can improve the performance of an application, using a simple example that counts the types of characters in some text data. The following example expands this concept, adding a different analysis routine on the second core.

The dual-core example can be found in this location:

```
Blackfin/Examples/No Hardware Required/
Compiler Features/Branch Prediction Dual-Core
```

The example project is called `Branch Prediction Dual-Core`. As a dual-core application, it also has a project group for the different sub-projects, all created with the Project Wizard. The example has a single `main.c` source file that contains the `main()` functions for both core A and core B.

As a result:

- Core A performs a word count analysis of the text, reporting the number of characters, words, and lines. (A word consists of any non-whitespace character sequence.) It also reports the cycle counts.
- Core B performs the type-of-character analysis seen in the single-core version of the example. It communicates its results to core A through global variables in common memory.

Since the same source file is compiled in two different ways to execute different algorithms, it cannot be optimized according to a single execution profile. Therefore, PGO session IDs are required.

To use the example, do the following:

1. Create a new IDDE simulator session for the ADSP-BF561 Blackfin processor.
2. Open the `Branch Prediction Dual-Core` project group.
3. Ensure that the Release configuration is selected.
4. In **Settings, Preferences**, ensure that the **Run to main** option is de-selected.
5. In **Project Options** for the `corea` sub-project, display the **Profile-Guided Optimization** page and select **Prepare application to create new profile** option.
6. Ensure that the **PGO session name** option is set to **CoreA**.
7. Do the same for the `coreb` sub-project, enabling the **Prepare application to create new profile** option and ensuring the **PGO session name option** is set to **CoreB**.
8. Rebuild everything and load the resulting `p0.dxe` into core A and `p1.dxe` into core B. They will be in the `Release` sub-directory.

9. Open a command-window, and change directory to the `system` sub-directory of the VisualDSP++ installation.

10. In the command-line window, execute `pgoctrl on` *`file.pgo`*, where *`file.pgo`* is a pathname to the file you'd like the profile to be stored in.

11. In the IDDE, execute a multi-core Run. You will have to do this a number of times (since each core halts at `_main`) until core A has reached `__lib_prog_term`. In the console window, core A will have reported the counts computed by each core and the cycles consumed by each while doing so.

12. In the command-line window, execute `pgoctrl off`. The IDDE will now create `file.pgo` where you specified.

13. In the **Project Options** for the `corea` and `coreb` sub-projects, clear the **Prepare application to create new profile** option and select the **Optimize using existing profiles** option. In the **Profile** field, browse to the `file.pgo` file just created.

14. Rebuild everything. The two versions of `main()` will now be rebuilt using the gathered profiles.

15. Reload the executables into cores A and B as before and run them until core A reaches `__lib_prog_term`.

16. Core A will report improved cycle counts for each core.

As for the single-core version of this example, the key decisions of each version of `main()` may also be predicted explicitly, using the `EXPRS` macro to select `expected_true()` or `expected_false()` branch prediction functions. See Figure 2-2 on page 2-36 for details.

Interprocedural Analysis and File Attributes

This example is in the `IPA Dual-Core` project group in the `IPA Dual-Core` directory. The example demonstrates how IPA can make dramatic improvements to an application, even in a dual-core system. The example uses file attributes for object placement.

Conflicting Approaches

The single/dual approach (on page A-16) uses sub-project libraries as an organizing mechanism. The dual-core `.ldf` file uses the libraries to control which application objects are mapped into particular regions of memory. However, this approach conflicts with a desire to use IPA: IPA propagates information about each source module and performs its analysis of all such source modules at link-time.

Where the analysis reveals some potential benefits, IPA recompiles the sources using the gathered information, and this is where the conflict arises. If IPA does not have sources available for recompilation, it cannot apply the benefits of the analysis. IPA can retrieve information from an object within a library, however, and can apply that information during analysis.

Therefore, to use IPA effectively in a dual-core environment, you have to ensure that any objects likely to benefit are linked directly into the application, and not via the conventional sub-project libraries.

Example Application

The example application performs a matrix operation. The `main()` function allocates a block of memory (using `getbuffer()`) to contain an NxM block of `short`s, and another array of N `short`s. It then calls another function `sumcol()` that sums the columns of the matrix into each element of the array:

```
array[i] += matrix[i][j];
```

Similar `main()` source is used for both cores, first allocating the memory and then counting the cycles required to perform the summing operation. The differences between the two cores are:

1. Different values for N and M are chosen for each core.
2. Core A contains additional code to enable core B, wait for core B to complete, and to display the cycle counts for both cores.
3. Core B contains additional code to pass its cycle count back to core A.

The same `getbuffer.c` source file is included in both the `corea` and `coreb` projects. The `main()` functions, with their differences as described, are in different source files, `maina.c` and `mainb.c`, in the `corea` and `coreb` projects. Because these files are part of the library sub-projects, they will not benefit from IPA's analysis, but can contribute to it.

The `sumcol()` function is handled differently.This example is arranged so that the `sumcol()` function is compiled separately for each core. Therefore the compiler can produce a version specialized for each core. If there was a single generic version, IPA would recognize that the functions were interacting in more than one fashion and would only be able to apply generic optimizations.

Building Multiple Instances of a Module

The function to be specialized by IPA is `sumcol()`, which is in the file `sumcol.h`. This is included into two further source files: `sumcola.c` and `sumcolb.c`, which are both part of the top-level project. For example, `maina.c` contains:

```
#define COREA
#pragma file_attr("DualCoreMem=CoreA")
#include "sumcol.h"
```

When the project is built, each of the two C source files will be built, producing two versions of each function, one per core. An alternative approach would be to build from the command-line (for example, using a makefile) and to specify different compiler options. For example:

```
ccblkfn -proc ADSP-BF561 sumcol.c -o sumcola.doj -multicore \
        -ipa -DCOREA -file-attr "DualCoreMem=CoreA"
ccblkfn -proc ADSP-BF561 sumcol.c -o sumcolb.doj -multicore \
        -ipa -DCOREB -file-attr "DualCoreMem=CoreB"
```

Since the IDDE does not support multiple builds of a single source module within a given project, the example uses the inclusion approach instead.

Libraries and File Attributes

The `.ldf` file used by the `IPA Dual-Core` project is generated from the Project Wizard, where the **LDF Settings, Multi-core Selection** is set to **Cores A and B**. This `.ldf` file uses the five-project convention and therefore expects to link against the four sub-project libraries. Therefore, these libraries exist here with the following contents:

- `corea` and `coreb` both contain a `main*.c` and `getbuffer.c`. This will cause the functions to be mapped into the private memory for each core. Since they placed into a library sub-project, IPA will have no effect on them, and they will not be specialized. However, IPA will record whatever information it can deduce about each, and will make that available during analysis.

- `sml2` contains `global.c`, which contains the global variables used to indicate core B's completion state and cycle count. It will be mapped into shared memory.

- `sml3` contains `dummy.c`. This library is not needed by the example, but the `.ldf` file expects it to exist.
- The top-level project contains the source files that are to be specialized by IPA: `sumcola.c` and `sumcolb.c`. It also contains source files auto-generated by **Project Options**, as part of the `.ldf` file production process.

Since the top-level project contains source files, these will be passed to the linker on the command-line. There must be some means by which the linker can determine how to map them into memory. This is achieved by the file attributes set in each source file. The `.ldf` file will map command-line objects into core A providing they do not have the file attribute `DualCoreMem=CoreB`. Similarly, it will map command-line objects into core B as along as they do not have the file attribute `DualCoreMem=CoreA`. This provides a mechanism for controlling file placement without placing the object files into specific libraries.

Multiple Definitions and Pragma Core

When an application contains multiple definitions of the same symbol and is being built using IPA, the IPA framework must distinguish between conflicting definitions. In this example, there are two definitions of `main()`, and two definitions of `sumcol()`. The definitions can be distinguished using `#pragma core` (on page 1-304). For example, the definition of `main()` in `maina.c` begins like this:

```
#pragma core("CoreA")
int main(void) {
```

When the function is compiled, the pragma will be used to specify different identifiers for the function. The same approach is used for the definition and declaration of `sumcol()`, except there, the pragma in question is compiled conditionally depending on the target core. During the IPA analysis, these identifiers allow the compiler to see that each version of `main()` is always calling a specific version of `sumcol()`, and therefore the compiler can propagate information about that call into the relevant version of `sumcol()`.

Note that each version of `main()` also calls `getbuffer()`, but this function does not need to be distinguished by `#pragma core` because its definition is being retrieved from a library. Therefore, it is not being specialized by IPA.

Since the top-level project contains auto-generated source files that do not have `#pragma core` on their definitions, these auto-generated files have file-specific options that do not include IPA.

Using the IPA Dual-Core Example

To use the IPA dual-core example, do the following:

1. Create a session in the VisualDSP++ IDDE for the ADSP-BF561 Blackfin processor.
2. Open the `IPA Dual-Core.dpg` project group.
3. Make `IPA Dual-Core` the active project.
4. Under **Project Options**, **Compile**, **General**, ensure that the **Interprocedural optimization** option is not enabled, but that the **Enable optimization** option is selected and that the slider is set to **100**.
5. Ensure the same settings apply for both `corea` and `coreb` projects.
6. Rebuild all.
7. Ensure that, when loading the resulting executables:
    - `P0.dxe` is loaded into core P0.
    - `P1.dxe` is loaded into core P1.

8. Run the two cores, until core A reaches `__lib_prog_term`. Core A will report the cycle counts for the two cores in the IDDE's console. This gives the counts for ordinary optimization without IPA.

9. To demonstrate the effect of IPA, select the **Interprocedural optimization** option under **Project Options**, **Compile**, **General**, for each of the top-level, `corea` and `coreb` projects.

10. Rebuild all.

11. Reload both executables into the two cores.

12. Rerun both cores. This time, the cycle counts will improve because IPA was able to propagate information about the parameters being passed from `main()` to `sumcol()`.

IPA Optimizations

There are several optimizations being done by IPA in this example:

- The values of N and M are propagated from parameters to values used within `sumcol()`, allowing the compiler to know the loop counts and memory access patterns.

- The memory allocated by `getbuffer()` is allocated by `malloc()`. It is known that all pointers returned by `malloc()`—and therefore by `getbuffer()`—will be optimally aligned and will be unique. (They will not alias other pointers returned in this context.) IPA then propagates this information to `sumcol()`.

- Recognizing the uniqueness, alignment and size of the allocated memory blocks allows the compiler to heavily optimize `sumcol()`, performing an unroll-and-jam transformation.

Where information about a library function is not available, because the library's objects were not built with IPA, it may be possible to explicitly announce information to the compiler. In this case, the `getbuffer()`

characteristics could be announced using `#pragma alloc` (on page 1-319) and `#pragma result_alignment` (on page 1-330).

Synchronization Functions

VisualDSP++ 5.0 provides functionality for synchronization. There are two compiler intrinsics (built-in functions) and three locking routines.

The compiler intrinsics are:

```
#include <ccblkfn.h>
int testset(char *);
void untestset(char *);
```

The `testset()` intrinsic generates a native `TESTSET` instruction, which can perform atomic updates on a memory location. The intrinsic returns the result of the `CC` flag produced by the `TESTSET` instruction. Refer to the *Blackfin Processor Programming Reference* for details.

The `untestset()` intrinsic clears the memory location set by the `testset()` intrinsic. This intrinsic is recommended in place of a normal memory write because the `untestset()` intrinsic acts as a stronger barrier to code movement during optimization.

The three locking routines are:

```
#include <ccblkfn.h>
void adi_acquire_lock(testset_t *);
int adi_try_lock(testset_t *);
void adi_release_lock(testset_t *);
```

The `adi_acquire_lock()` routine repeatedly attempts to claim the lock by issuing `testset()` until successful, whereupon it returns to the caller. In contrast, the `adi_try_lock()` routine makes a single attempt—if it successfully claims the lock, it returns nonzero; otherwise, it returns zero.

The adi_release_lock() routine releases the lock obtained by either adi_acquire_lock() or adi_try_lock(). It assumes that the lock was already claimed and makes no attempt to verify that its caller is in fact the current owner of the lock. None of these intrinsics or functions disable interrupts—that is left to the caller's discretion.

I INDEX

Numerics

A

D

F

O

S

T

U

X

Z