

VISUALDSP++™ 3.0

C/C++ Compiler and Library

Manual for Blackfin™ DSPs

Second Revision, April 2002

Part Number
82-000410-03

Analog Devices, Inc.
Digital Signal Processor Division
One Technology Way
Norwood, Mass. 02062-9106



Copyright Information

©2002 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

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

Trademark and Service Mark Notice

The Analog Devices logo, VisualDSP, the VisualDSP logo, SHARC, the SHARC logo, TigerSHARC, and the TigerSHARC logo are registered trademarks of Analog Devices, Inc.

VisualDSP++, the VisualDSP++ logo, CROSSCORE, the CROSSCORE logo, Blackfin, the Blackfin logo, and EZ-KIT Lite are trademarks of Analog Devices, Inc.

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

CONTENTS

PREFACE

| | |
|--------------------------------------|--------|
| Purpose | xxiii |
| Intended Audience | xxiii |
| Manual Contents Description | xxiv |
| What's New in this Manual | xxiv |
| Technical or Customer Support | xxv |
| Supported Processors | xxv |
| Product Information | xxvi |
| MyAnalog.com | xxvi |
| DSP Product Information | xxvi |
| Related Documents | xxvii |
| Online Technical Documentation | xxviii |
| From VisualDSP++ | xxviii |
| From Windows | xxix |
| From the Web | xxix |
| Printed Manuals | xxx |
| VisualDSP++ Documentation Set | xxx |
| Hardware Manuals | xxx |
| Datasheets | xxx |

CONTENTS

| | |
|-----------------------------------|------|
| Contacting DSP Publications | xxxi |
| Notation Conventions | xxxi |

COMPILER

| | |
|--|------|
| C/C++ Compiler Overview | 1-2 |
| Compiler Command-Line Interface | 1-4 |
| Running the Compiler | 1-5 |
| Specifying Compiler Options in VisualDSP++ | 1-9 |
| C/C++ Compiler Switches | 1-10 |
| C/C++ Compiler Switch Summaries | 1-10 |
| C/C++ Mode Selection Switch Descriptions | 1-20 |
| -analog | 1-20 |
| -c++ | 1-20 |
| -traditional | 1-20 |
| C/C++ Compiler Common Switch Descriptions | 1-21 |
| sourcefile | 1-21 |
| -@ filename | 1-21 |
| -21352, -21535 | 1-21 |
| -A name[tokens] | 1-21 |
| -alttok | 1-22 |
| -auto-inline factor | 1-22 |
| -BLACKFIN | 1-23 |
| -build-lib | 1-23 |
| -C | 1-23 |
| -c | 1-24 |

| | |
|--|------|
| -circbuf | 1-24 |
| -const-read-write | 1-24 |
| -debug-types | 1-24 |
| -Dmacro[=definition] | 1-25 |
| -dry | 1-25 |
| -dryrun | 1-25 |
| -E | 1-25 |
| -EE | 1-26 |
| -extra-keywords | 1-26 |
| -flags{-asm -compiler -lib -link} switch [,switch2 [...]] | 1-26 |
| -full-version | 1-26 |
| -g | 1-26 |
| -H | 1-27 |
| -HH | 1-27 |
| -h[elp] | 1-27 |
| -I directory [{, ;} directory...] | 1-28 |
| -include filename | 1-28 |
| -ipa | 1-28 |
| -jcs2l | 1-28 |
| -L directory[{, ;} directory...] | 1-28 |
| -l library | 1-28 |
| -M | 1-29 |
| -MM | 1-29 |
| -Mt filename | 1-29 |

CONTENTS

| | |
|---|------|
| -map filename | 1-30 |
| -no-alttok | 1-30 |
| -no-builtin | 1-30 |
| -no-defs | 1-30 |
| -no-extra-keywords | 1-30 |
| -no-inline | 1-31 |
| -no-int-to-fract | 1-31 |
| -no-jcs2l | 1-31 |
| -no-restrict | 1-31 |
| -no-std-def | 1-31 |
| -no-std-inc | 1-32 |
| -no-std-lib | 1-32 |
| -nothreads | 1-32 |
| -O | 1-32 |
| -Ofp | 1-33 |
| -Os | 1-33 |
| -Ox | 1-33 |
| -Oz | 1-34 |
| --o filename | 1-34 |
| -P | 1-34 |
| -PP | 1-34 |
| -p[1 2] | 1-35 |
| -path [-asm -compiler -lib -link] directory | 1-35 |
| -path-install directory | 1-35 |

| | |
|--|------|
| -path-output directory | 1-35 |
| -path-temp directory | 1-36 |
| -pedantic | 1-36 |
| -pedantic-errors | 1-36 |
| -pplist filename | 1-36 |
| -proc processorID | 1-37 |
| -R directory[{: ,}directory ...] | 1-37 |
| -restrict | 1-37 |
| -S | 1-38 |
| -s | 1-38 |
| -sat32 | 1-38 |
| -sat40 | 1-38 |
| -save-temps | 1-38 |
| -show | 1-39 |
| -signed-char | 1-39 |
| -syntax-only | 1-39 |
| -T filename | 1-39 |
| -threads | 1-39 |
| -time | 1-40 |
| -Umacro | 1-40 |
| -unsigned-char | 1-40 |
| -v | 1-40 |
| -verbose | 1-40 |
| -version | 1-40 |

CONTENTS

| | |
|---|------|
| -warn-protos | 1-41 |
| -W[error remark suppress warn] number[, number ...] ... | 1-41 |
| -Wdriver-limit number | 1-41 |
| -Werror-limit number | 1-41 |
| -Wremarks | 1-41 |
| -Wterse | 1-42 |
| -w | 1-42 |
| -write-files | 1-42 |
| -xref filename | 1-42 |
| C++ Mode Compiler Switch Descriptions | 1-43 |
| -explicit | 1-43 |
| -instant[all used] | 1-43 |
| -namespace | 1-44 |
| -newforinit | 1-44 |
| -newvec | 1-44 |
| -no-demangle | 1-44 |
| -no-explicit | 1-44 |
| -no-namespace | 1-44 |
| -no-newvec | 1-45 |
| -notstrict | 1-45 |
| -no-wchar | 1-45 |
| -strict | 1-45 |
| -strictwarn | 1-45 |
| -tpautooff | 1-46 |

| | |
|--|------|
| -trdforinit | 1-46 |
| -typename | 1-46 |
| -wchar | 1-46 |
| Data Type Sizes | 1-47 |
| Optimization Control | 1-48 |
| Inlining Control | 1-51 |
| Interprocedural Analysis | 1-52 |
| Interaction with Libraries | 1-52 |
| C/C++ Compiler Language Extensions | 1-55 |
| Inline Function Support Keyword (inline) | 1-57 |
| Inline Assembly Language Support Keyword (asm) | 1-58 |
| Assembly Construct Template | 1-59 |
| asm() Constructs Syntax | 1-59 |
| asm() Construct Syntax Rules | 1-60 |
| asm() Construct Template Example | 1-61 |
| Assembly Construct Operand Description | 1-62 |
| Assembly Constructs with Multiple Instructions | 1-68 |
| Assembly Construct Reordering and Optimization | 1-68 |
| Assembly Constructs with Input and Output Operands | 1-69 |
| Assembly Constructs and Flow Control | 1-70 |
| Placement Support Keyword (section) | 1-70 |
| Boolean Type Support Keywords (bool, true, false) | 1-71 |
| Pointer Class Support Keyword (restrict) | 1-71 |
| Non-Constant Aggregate Initializer Support | 1-72 |

CONTENTS

| | |
|--|------|
| Indexed Initializer Support | 1-73 |
| Preprocessor Generated Warnings | 1-75 |
| Variable Length Arrays | 1-75 |
| C++ Style Comments | 1-75 |
| Built-In Functions | 1-76 |
| Fractional Value Built-Ins | 1-76 |
| Single Fractional Values | 1-77 |
| Complex Fractional Built-ins | 1-79 |
| Viterbi History and Decoding Functions | 1-80 |
| Non-Fractional Arithmetic Built-ins | 1-82 |
| Circular Buffer Built-In Functions | 1-83 |
| System Built-In Functions | 1-84 |
| Pragmas | 1-85 |
| Data Alignment Pragmas | 1-86 |
| Interrupt Handler Pragmas | 1-88 |
| Loop Optimization | 1-88 |
| General Optimization | 1-89 |
| Linking Pragma Behavior | 1-90 |
| Pragmas to Assist in Vectorizing Loops | 1-90 |
| Blackfin DSP Specific Functionality | 1-93 |
| Default Startup Code | 1-93 |
| Support for argv/argc | 1-93 |
| File I/O Support | 1-94 |
| Profiling with Instrumented Code | 1-95 |

| | |
|---|-------|
| Generating Instrumented Code | 1-96 |
| Linking the Executable | 1-96 |
| Running the Executable | 1-97 |
| Post-Processing mon.out File | 1-97 |
| Computing Cycle Counts | 1-98 |
| Controlling Available Memory Size | 1-98 |
| Interrupt Handler Support | 1-99 |
| Defining an ISR | 1-100 |
| Registering an ISR | 1-101 |
| ISRs and ANSI C Signals | 1-102 |
| Saved Processor Context | 1-102 |
| Fetching Event Details | 1-103 |
| Fetching Saved Registers | 1-104 |
| User-Mode Configuration | 1-105 |
| Allocated Events in User-Mode Configuration | 1-105 |
| C/C++ Preprocessor Features | 1-107 |
| Predefined Macros | 1-107 |
| Preprocessing of .IDL Files | 1-109 |
| Header Files | 1-110 |
| Writing Preprocessor Macros | 1-110 |
| C/C++ Run-Time Model and Environment | 1-113 |
| Using Memory Sections | 1-114 |
| Dedicated Registers | 1-116 |
| Call Preserved Registers | 1-116 |

CONTENTS

| | |
|---|-------|
| Scratch Registers | 1-117 |
| Stack Registers | 1-117 |
| Managing the Stack | 1-117 |
| Transferring Function Arguments and Return Value | 1-121 |
| Passing Arguments | 1-121 |
| Return Values | 1-122 |
| Using Data Storage Formats | 1-124 |
| Basic Startup Code Sequence | 1-126 |
| C/C++ and Assembly Interface | 1-128 |
| Calling Assembly Subroutines from C/C++ Programs | 1-128 |
| Calling C/C++ Functions from Assembly Programs | 1-130 |
| Using Mixed C/C++ and Assembly Naming Conventions | 1-131 |
| C++ Programming Examples | 1-133 |
| Using Fract Support | 1-133 |
| Using Complex Support | 1-134 |
| C/C++ RUN-TIME LIBRARY | |
| C and C++ Run-Time Library Guide | 2-3 |
| Calling Library Functions | 2-3 |
| Using the Compiler's Built-In Functions | 2-4 |
| Linking Library Functions | 2-4 |
| Working with Library Header Files | 2-8 |
| assert.h | 2-9 |
| ctype.h | 2-9 |
| errno.h | 2-9 |

| | |
|---|------|
| float.h | 2-9 |
| limits.h | 2-10 |
| locale.h | 2-10 |
| math.h | 2-10 |
| setjmp.h | 2-11 |
| signal.h | 2-11 |
| stdarg.h | 2-11 |
| stddef.h | 2-11 |
| stdio.h | 2-11 |
| stdlib.h | 2-12 |
| string.h | 2-12 |
| Abridged C++ Library Support | 2-13 |
| Embedded C++ Library Header Files | 2-13 |
| complex | 2-13 |
| exception | 2-13 |
| fstream | 2-14 |
| iomanip | 2-14 |
| ios | 2-14 |
| iosfwd | 2-14 |
| iostream | 2-14 |
| istream | 2-14 |
| new | 2-14 |
| ostream | 2-15 |
| sstream | 2-15 |

CONTENTS

| | |
|---|------|
| stdexcept | 2-15 |
| streambuf | 2-15 |
| string | 2-15 |
| strstream | 2-15 |
| C++ Header Files for C Library Facilities | 2-15 |
| Embedded Standard Template Library Header Files | 2-17 |
| algorithm | 2-17 |
| deque | 2-17 |
| functional | 2-17 |
| hash_map | 2-17 |
| hash_set | 2-17 |
| iterator | 2-17 |
| list | 2-17 |
| map | 2-17 |
| memory | 2-18 |
| numeric | 2-18 |
| queue | 2-18 |
| set | 2-18 |
| stack | 2-18 |
| utility | 2-18 |
| vector | 2-18 |
| fstream.h | 2-18 |
| iomanip.h | 2-18 |
| iostream.h | 2-19 |

| | |
|------------------------------------|------|
| new.h | 2-19 |
| Documented Library Functions | 2-20 |
| C Run-Time Library Reference | 2-23 |
| Notation Conventions. | 2-23 |
| abort | 2-24 |
| abs | 2-25 |
| acos | 2-26 |
| asin | 2-27 |
| atan | 2-28 |
| atan2 | 2-29 |
| atexit | 2-30 |
| atof | 2-31 |
| atoi | 2-32 |
| atol | 2-33 |
| bsearch | 2-34 |
| calloc | 2-36 |
| ceil | 2-37 |
| cos | 2-38 |
| cosh | 2-39 |
| div | 2-40 |
| exit | 2-41 |
| exp | 2-42 |
| fabs | 2-43 |
| floor | 2-44 |

CONTENTS

| | |
|-----------------|------|
| fmod | 2-45 |
| free | 2-46 |
| frexp | 2-47 |
| interrupt | 2-48 |
| isalnum | 2-49 |
| isalpha | 2-50 |
| iscntrl | 2-51 |
| isdigit | 2-52 |
| isgraph | 2-53 |
| islower | 2-54 |
| isprint | 2-55 |
| ispunct | 2-56 |
| isspace | 2-57 |
| isupper | 2-58 |
| isxdigit | 2-59 |
| labs | 2-60 |
| ldexp | 2-61 |
| ldiv | 2-62 |
| log | 2-63 |
| log10 | 2-64 |
| longjmp | 2-65 |
| malloc | 2-67 |
| memchr | 2-68 |
| memcmp | 2-69 |

| | |
|----------------|------|
| memcpy | 2-70 |
| memmove | 2-71 |
| memset | 2-72 |
| modf | 2-73 |
| pow | 2-74 |
| qsort | 2-75 |
| raise | 2-77 |
| rand | 2-79 |
| realloc | 2-80 |
| setjmp | 2-81 |
| signal | 2-82 |
| sin | 2-83 |
| sinh | 2-84 |
| sqrt | 2-85 |
| srand | 2-86 |
| strcat | 2-87 |
| strchr | 2-88 |
| strcmp | 2-89 |
| strcoll | 2-90 |
| strcpy | 2-91 |
| strcspn | 2-92 |
| strerror | 2-93 |
| strlen | 2-94 |
| strncat | 2-95 |

CONTENTS

| | |
|----------------|-------|
| strncmp | 2-96 |
| strncpy | 2-97 |
| strpbrk | 2-98 |
| strchr | 2-99 |
| strspn | 2-100 |
| strstr | 2-101 |
| strtod | 2-102 |
| strtok | 2-104 |
| strtol | 2-106 |
| strtoul | 2-108 |
| strxfrm | 2-110 |
| tan | 2-112 |
| tanh | 2-113 |
| tolower | 2-114 |
| toupper | 2-115 |
| va_arg | 2-116 |
| va_end | 2-118 |
| va_start | 2-119 |

DSP RUN-TIME LIBRARY

| | |
|--|-----|
| DSP Run-Time Library Guide | 3-2 |
| Linking DSP Library Functions | 3-2 |
| Working With Library Source Code | 3-2 |
| DSP Header Files | 3-3 |
| complex.h — Basic Complex Arithmetic Functions | 3-4 |

| | |
|--|------|
| filter.h — DSP Filters and Transformations | 3-4 |
| math.h — Math Functions | 3-10 |
| matrix.h — Matrix Functions | 3-12 |
| stats.h — Statistical Functions | 3-16 |
| vector.h — Vector Functions | 3-18 |
| window.h — Window Generators | 3-18 |
| DSP Run-Time Library Reference | 3-25 |
| Notation Conventions | 3-25 |
| a_compress | 3-26 |
| a_expand | 3-27 |
| arg | 3-28 |
| autocoh | 3-29 |
| autocorr | 3-30 |
| cabs | 3-31 |
| cadd | 3-32 |
| cdiv | 3-33 |
| cexp | 3-34 |
| cfft | 3-35 |
| cfft2d | 3-37 |
| cfft2d | 3-39 |
| cfir | 3-41 |
| clip | 3-43 |
| cmlt | 3-44 |
| conj | 3-45 |

CONTENTS

| | |
|-----------------------|------|
| convolve | 3-46 |
| conv2d | 3-48 |
| conv2d3x3 | 3-50 |
| copysign | 3-51 |
| cot | 3-52 |
| countones | 3-53 |
| crosscoh | 3-54 |
| crosscorr | 3-56 |
| csub | 3-57 |
| fir | 3-58 |
| fir_decima | 3-60 |
| fir_interp | 3-62 |
| gen_bartlett | 3-64 |
| gen_blackman | 3-65 |
| gen_gaussian | 3-66 |
| gen_hamming | 3-67 |
| gen_hanning | 3-68 |
| gen_harris | 3-69 |
| gen_kaiser | 3-70 |
| gen_rectangular | 3-71 |
| gen_triangle | 3-72 |
| gen_vonhann | 3-74 |
| histogram | 3-75 |
| ifft | 3-76 |

| | |
|--------------------|-------|
| iffttrad4 | 3-78 |
| ifft2d | 3-80 |
| iir | 3-82 |
| max | 3-84 |
| mean | 3-85 |
| min | 3-86 |
| mu_compress | 3-87 |
| mu_expand | 3-88 |
| norm | 3-89 |
| polar | 3-90 |
| rfft | 3-91 |
| rffttrad4 | 3-93 |
| rfft2d | 3-95 |
| rms | 3-97 |
| rsqrt | 3-98 |
| twidffttrad2 | 3-99 |
| twidffttrad4 | 3-101 |
| twidfft2d | 3-103 |
| var | 3-105 |
| zero_cross | 3-106 |

INDEX

CONTENTS

PREFACE

Thank you for purchasing Analog Devices development software for digital signal processors (DSPs).

Purpose

The *VisualDSP++ 3.0 C/C++ Compiler and Library Manual for Blackfin DSPs* contains information about the C/C++ compiler and run-time library program for Blackfin™ DSPs. It includes syntax for command lines, switches, and language extensions. It leads you through the process of using library routines and writing mixed C/C++/assembly code.

Intended Audience

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

Programmers who are unfamiliar with Blackfin DSPs can use this manual, but they should supplement it with other texts (such as the appropriate hardware reference and instruction set reference) that provide information about your Blackfin DSP architecture and instructions).

Manual Contents Description

This manual contains:

- Chapter 1, “Compiler”
Provides information on compiler options, language extensions and C/C++/assembly interfacing
- Chapter 2, “C/C++ Run-Time Library”
Shows how to use library functions and provides a complete C/C++ library function reference
- Chapter 3, “DSP Run-Time Library”
Shows how to use DSP library functions and provides a complete DSP library function reference

What’s New in this Manual

This edition of the *VisualDSP++ 3.0 C/C++ Compiler and Library Manual for Blackfin DSPs* documents support for all Blackfin processors.

In addition to documenting all existing compiler features, this manual describes new features including: optimization pragmas, circular buffer intrinsics, source annotations, complex maths functions, IDL file preprocessing, and Blackfin-specific enhancements.

Technical or Customer Support

You can reach DSP Tools Support in the following ways:

- Visit the DSP Development Tools website at
<http://www.analog.com/technology/dsp/development-Tools/index.html>
- Email questions to
dsptools.support@analog.com
- Phone questions to 1-800-ANALOGD
- Contact your ADI local sales office or authorized distributor
- Send questions by mail to:

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

Supported Processors

The name “*Blackfin*” refers to a family of Analog Devices 16-bit, fixed-point processors. VisualDSP++ currently supports the following Blackfin processors:

- ADSP-21532 DSP
- ADSP-21535 DSP

Product Information

You can obtain product information from the Analog Devices website, from the product CD-ROM, or from the printed publications (manuals).

Analog Devices is online at www.analog.com. Our website provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

MyAnalog.com

MyAnalog.com is a free feature of the Analog Devices website that allows customization of a webpage to display only the latest information on products you are interested in. You can also choose to receive weekly email notification containing updates to the webpages that meet your interests. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Registration:

Visit www.myanalog.com to sign up. Click **Register** to use MyAnalog.com. Registration takes about five minutes and serves as means for you to select the information you want to receive.

If you are already a registered user, just log on. Your user name is your email address.

DSP Product Information

For information on digital signal processors, visit our website at www.analog.com/dsp, which provides access to technical publications, datasheets, application notes, product overviews, and product announcements.

You may also obtain additional information about Analog Devices and its products in any of the following ways.

- Email questions or requests for information to dsp.support@analog.com
- Fax questions or requests for information to
1-781-461-3010 (North America)
089/76 903-557 (Europe)
- Access the Digital Signal Processing Division's FTP website at
[ftp ftp.analog.com](ftp://ftp.analog.com) or **ftp 137.71.23.21**
<ftp://ftp.analog.com>

Related Documents

For information on product related development software, see the following publications:

VisualDSP++ 3.0 Getting Started Guide for Blackfin DSPs

VisualDSP++ 3.0 User's Guide for Blackfin DSPs

VisualDSP++ 3.0 C/C++ Compiler and Library Manual for Blackfin DSPs

VisualDSP++ 3.0 C/C++ Assembler and Preprocessor Manual for Blackfin DSPs

VisualDSP++ 3.0 Linker and Utilities Manual for Blackfin DSPs

VisualDSP++ 3.0 Product Bulletin

VisualDSP++ Kernel (VDK) User's Guide

VisualDSP++ Component Software Engineering User's Guide

Quick Installation Reference Card

Online Technical Documentation

Online documentation comprises VisualDSP++ Help system and tools manuals, Dinkum Abridged C++ library and FlexLM network license manager software documentation. You can easily search across the entire VisualDSP++ documentation set for any topic of interest. For easy printing, supplementary .PDF files for the tools manuals are also provided.

A description of each documentation file type is as follows.

| File | Description |
|-------|--|
| .CHM | Help system files and VisualDSP++ tools manuals. |
| .HTML | Dinkum Abridged C++ library and FlexLM network license manager software documentation. Viewing and printing the .HTML files require a browser, such as Internet Explorer 4.0 (or higher). |
| .PDF | VisualDSP++ tools manuals in Portable Documentation Format, one .PDF file for each manual. Viewing and printing the .PDF files require a PDF reader, such as Adobe Acrobat Reader (4.0 or higher). |

If documentation is not installed on your system as part of the software installation, you can add it from the VisualDSP++ CD-ROM at any time.

Access the online documentation from the VisualDSP++ environment, Windows Explorer, or Analog Devices website.

From VisualDSP++

- Access VisualDSP++ online Help from the Help menu's **Contents**, **Search**, and **Index** commands.
- Open online Help from context-sensitive user interface items (tool-bar buttons, menu commands, and windows).

From Windows

In addition to any shortcuts you may have constructed, there are many ways to open VisualDSP++ online Help or the supplementary documentation from Windows.

Help system files (.CHM files) are located in the `Help` folder, and .PDF files are located in the `Docs` folder of your VisualDSP++ installation. The `Docs` folder also contains the Dinkum Abridged C++ library and FlexLM network license manager software documentation.

Using Windows Explorer

- Double-click any file that is part of the VisualDSP++ documentation set.
- Double-click the `vdsp-help.chm` file, which is the master Help system, to access all the other .CHM files.

Using the Windows Start Button

- Access VisualDSP++ online Help by clicking the **Start** button and choosing **Programs, VisualDSP, and VisualDSP++ Documentation**.
- Access the .PDF files by clicking the **Start** button and choosing **Programs, VisualDSP, Documentation for Printing**, and the name of the book.

From the Web

To download the tools manuals, point your browser at http://www.analog.com/technology/dsp/developmentTools/gen_purpose.html.

Select a DSP family and book title. Download archive (.ZIP) files, one for each manual. Use any archive management software, such as WinZip, to decompress downloaded files.

Product Information

Printed Manuals

For general questions regarding literature ordering, call the Literature Center at **1-800-ANALOGD (1-800-262-5643)** and follow the prompts.

VisualDSP++ Documentation Set

VisualDSP++ manuals may be purchased through Analog Devices Customer Service at **1-781-329-4700**; ask for a Customer Service representative. The manuals can be purchased only as a kit. For additional information, call **1-603-883-2430**.

If you do not have an account with Analog Devices, you will be referred to Analog Devices distributors. To get information on our distributors, log onto <http://www.analog.com/salesdir/continent.asp>.

Hardware Manuals

Hardware reference and instruction set reference manuals can be ordered through the Literature Center or downloaded from the Analog Devices website. The phone number is **1-800-ANALOGD (1-800-262-5643)**. The manuals can be ordered by a title or by product number located on the back cover of each manual.

Datasheets

All datasheets can be downloaded from the Analog Devices website. As a general rule, any datasheet with a letter suffix (L, M, N) can be obtained from the Literature Center at **1-800-ANALOGD (1-800-262-5643)** or downloaded from the website. Datasheets without the suffix can be downloaded from the website only—no hard copies are available. You can ask for the datasheet by a part name or by product number.

If you want to have a datasheet faxed to you, the fax number for that service is **1-800-446-6212**. Follow the prompts and a list of datasheet code numbers will be faxed to you. Call the Literature Center first to find out if requested datasheets are available.


Contacting DSP Publications

Please send your comments and recommendation on how to improve our manuals and online Help. You can contact us by:

- Emailing `dsp.techpubs@analog.com`
- Filling in and returning the attached Reader's Comments Card found in our manuals



Notation Conventions

The following table identifies and describes text conventions used in this manual.

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

| Example | Description |
|-------------------------------------|--|
| Close command (File menu) | Text in bold style indicates 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 <i>this</i> or <i>that</i> . |
| [this that] | Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional <i>this</i> or <i>that</i> . |

Notation Conventions

| Example | Description |
|---|---|
| [this,...] | Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipsis; read the example as an optional comma-separated list of <i>this</i> . |
| .SECTION | Commands, directives, keywords, and feature names are in text with letter gothic font. |
| <i>filename</i> | Non-keyword placeholders appear in text with italic style format. |
|  | A note, providing information of special interest or identifying a related topic. In the online version of this book, the word Note appears instead of this symbol. |
|  | A caution, providing information about critical design or programming issues that influence operation of a product. In the online version of this book, the word Caution appears instead of this symbol. |

1 COMPILER

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

This chapter contains:

- [“C/C++ Compiler Overview” on page 1-2](#)
Provides an overview of C/C++ compiler for Blackfin DSPs.
- [“Compiler Command-Line Interface” on page 1-4](#)
Describes the operation of the compiler as it processes programs, including input and output files and command-line switches.
- [“C/C++ Compiler Language Extensions” on page 1-55](#)
Describes the `ccblkfn` compiler’s extensions to the ANSI/ISO standard for the C and C++ languages.
- [“C/C++ Preprocessor Features” on page 1-107](#)
Contains information on the preprocessor and ways to modify source compilation.
- [“C/C++ Run-Time Model and Environment” on page 1-113](#)
Contains reference information about implementation of C/C++ programs, data, and function calls in Blackfin DSPs.
- [“C/C++ and Assembly Interface” on page 1-128](#)
which describes how to call an assembly language subroutine from within a C or C++ program, and how to call a C or C++ function from within an assembly language program.

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 DSP operations without having to understand the underlying DSP architecture.

The C/C++ compiler compiles ANSI/ISO standard C and C++ code to support DSP-based 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 an 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 DSP family assembler (`easmbkfn.exe`). The assembler creates Executable and Linkable Format (ELF) object files that can be linked (using the linker) to create a Blackfin DSP executable file or included in an archive library (`elfar.exe`). 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.

Source files contain the C/C++ program to be processed by the compiler. The `ccblkfn` compiler supports the ANSI/ISO standard definitions of the C and C++ languages. For information on 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 `ccblkf` compiler supports the proposed Embedded C++ Standard, which defines a subset of the full ISO/IEC 14882:1998 C++ language standard. The proposal excludes features that can detract from compiler performance in embedded systems, such as exception handling and run-time type identification. In addition to the embedded C++ standard features, `ccblkf` supports templates and all features of the full C++ standard except for the exception handling and run-time type identifications. The additional supported features provide extra functionality without degrading compiler performance.

The `ccblkf` compiler supports a set of C/C++ language extensions. These extensions support hardware features of the Blackfin DSPs. For information on these extensions, see “[C/C++ Compiler Language Extensions](#)” on [page 1-55](#).

You can set the compiler options from the **Compile** page of the **Project Options** dialog box of the VisualDSP++ 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, see the *VisualDSP++ 3.0 User’s Guide for Blackfin DSPs* and online Help.

Compiler Command-Line Interface

This section describes how the `ccblkfnc` 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 1-5](#)
- [“Specifying Compiler Options in VisualDSP++” on page 1-9](#)
- [“C/C++ Compiler Switches” on page 1-10](#)
- [“Data Type Sizes” on page 1-47](#)
- [“Optimization Control” on page 1-48](#)

By default, the compiler runs in C mode. This means that the compiler processes source files written in ANSI/ISO standard C language supplemented with Analog Devices extensions. Several options allow you to change the compilation mode and language dialect, thus enforcing certain standards and/or disabling the Analog Devices extensions. [Table 1-1 on page 1-8](#) lists valid extensions. By default, the compiler processes the input file through the listed stages to produce a `.DXX` file. [Table 1-2 on page 1-10](#) lists the 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 1-3 on page 1-11](#). A summary of the C++ specific compiler switches appears in [Table 1-4 on page 1-18](#). The summaries are followed by descriptions of each switch.



When developing a DSP project, you may find it useful to modify the compiler's default options settings. The way you set the compiler's options depends on the environment used to run the DSP development software. See [“Specifying Compiler Options in VisualDSP++” on page 1-9](#) for more information.

Running the Compiler

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

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

where:

- `-switch` is the name of the switch to be processed. The compiler has many switches. These 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` is the name of the file to be preprocessed, compiled, assembled, and/or linked.

A file name can include the drive, directory, file name and file extension. The compiler supports both Win32 and POSIX-style paths by using forward or back slashes as the directory delimiter. It also supports UNC path names (starting with two slashes and a network name).

If a file name length exceeds eight characters or contains spaces, enclose it in straight quotes; for example, "long file name.c". The `ccblkfn` compiler uses the file extension to determine what the file contains and what operations to perform upon it. [Table 1-1 on page 1-8](#) lists valid extensions.

For example, the following command line

```
ccblkfn -O -Wremarks -o program.dxe source.c
```

runs `ccblkfn` with

| | |
|------------------------|--|
| <code>-O</code> | Specifies optimization for the compiler |
| <code>-Wremarks</code> | Selects extra diagnostic remarks in addition to warning and error messages |

Compiler Command-Line Interface

| | |
|-----------------------------|---|
| <code>-o program.dxe</code> | Selects a name for the compiled, linked output |
| <code>source.c</code> | Specifies the C language source file to be compiled |

The following example command line, which runs the compiler in the C++ mode,

```
ccblkfn -c++ source.cpp
```

runs `ccblkfn` with

| | |
|-------------------------|---|
| <code>-c++</code> | Specifies all of the source files be compiled in C++ mode |
| <code>source.cpp</code> | 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 following file extensions to determine the actions to be performed.

| Extension | Action |
|---|---|
| <code>.c</code> , <code>.cpp</code> , <code>.cxx</code> | Source file is compiled, assembled, and linked |
| <code>.asm</code> or <code>.s</code> | Assembly language source file is assembled and linked |
| <code>.obj</code> | Object file (from previous assembly) is linked |

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, or selecting options with the VisualDSP++ IDDE. These switches are `-E`, `-P`, `-M`, `-H`, `-S`, `-c`.

Many of the compiler's switches take a file name as an optional parameter. If you do not use the optional output name switch, `ccb1kfn` names the output for you. [Table 1-1 on page 1-8](#) lists the type of files, names, and extensions `ccb1kfn` 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, search directories that you select, and path information that you include in the file name. [Table 1-1](#) indicates the extensions that the preprocessor, compiler, assembler, and linker support. The compiler supports relative and absolute directory names to define file extensions paths. For information on additional search directories, see the command-line switch that controls the specific type of extensions.

When you provide an input or output file name as an optional parameter, use the following guidelines.

- Use a file name (include the file extension) with either an unambiguous relative path or an absolute path. A file name with an absolute path includes the drive, directory, file name, and file extension.

Enclose long file names within straight quotes; for example, "long file name.c". The compiler uses the file extension convention listed in [Table 1-1](#) to determine the input file type.

- Verify 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, `ccb1kfn` looks for input in the current directory.



Using the verbose output switches for the preprocessor, compiler, assembler, and linker cause each of these tools to echo the name of each file as it is processed.

Compiler Command-Line Interface

Table 1-1. Input and Output Files

| File Extension | File Extension Description |
|----------------|--|
| .c | C source file |
| .cpp, .cxx | C++ source code |
| .h | Header file (referenced by an <code>#include</code> statement) |
| .ii, .ti | Template instantiation files — used internally by the compiler when instantiating templates |
| .ipa, .opa | Interprocedural analysis files — used internally by the compiler when performing interprocedural analysis. |
| .i | Preprocessed source file — created when preprocess only is specified |
| .s, .asm | Assembly language source files |
| .is | Preprocessed assembly language source — retained when <code>-save-temps</code> is specified |
| .ldf | Linker Description File |
| .obj | Object file to be linked |
| .dlb | Library of object files to be linked as needed |
| .dx | Executable file produced by compiler |
| .map | DSP system memory map file output |
| .sym | DSP system symbol map file output |

Specifying Compiler Options in VisualDSP++

When using the VisualDSP++ IDDE, use the **Compile** page from the **Project Options** dialog box to set compiler functional options. Callouts refer to the corresponding compiler command-line switches described in “[C/C++ Compiler Switches](#)”. The **Additional options** field is used to enter the appropriate file names and options that do not have corresponding controls on the **Compile** page but are available as compiler switches.

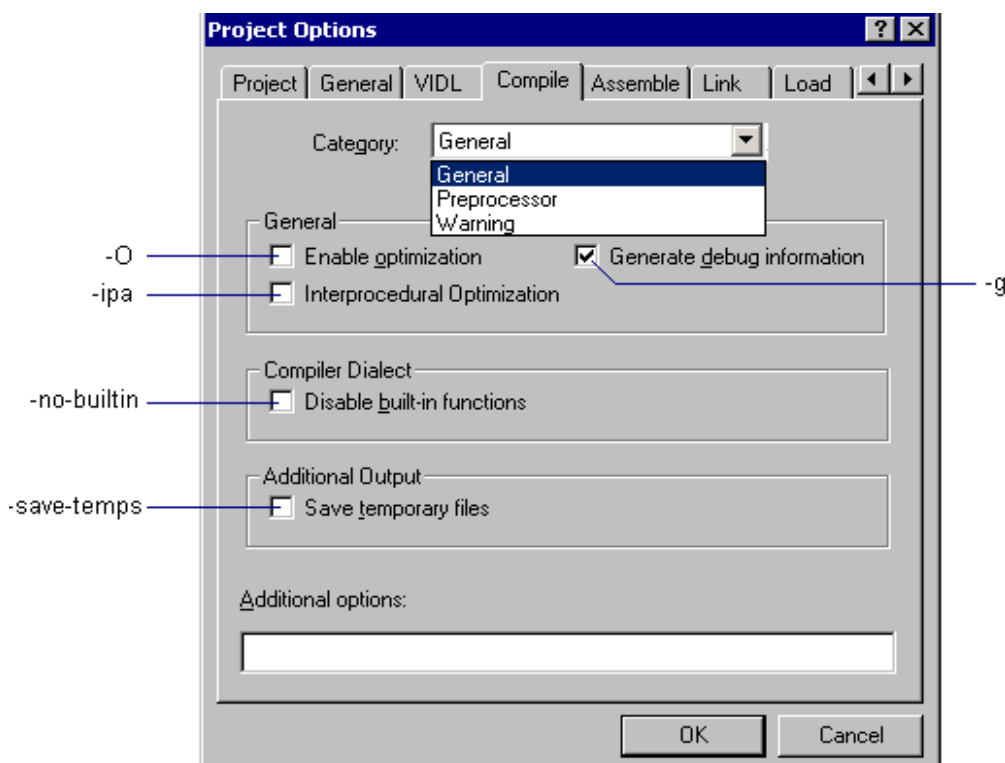


Figure 1-1. Project Options – Compile Property Page

Use the VisualDSP++ online Help to get more information on compiler options you can specify from the VisualDSP++ environment.

C/C++ Compiler Switches

This section describes command-line switches you can use when compiling. It contains a set of tables that provides a brief description of each switch. These tables are organized by type of switch. Following these tables are sections that provide detailed switch descriptions.

C/C++ Compiler Switch Summaries

This section contains a set of tables that summarize generic and specific switches (options).

- [Table 1-2, “C or C++ Mode Selection Switches” on page 1-10](#)
- [Table 1-3, “C/C++ Compiler Common Switches” on page 1-11](#)
- [Table 1-4, “C++ Mode Compiler Switches” on page 1-18](#)

A brief description of each switch appears in the sections beginning [on page 1-20](#).

Table 1-2. C or C++ Mode Selection Switches

| Switch Name | Description |
|--------------|--|
| -analog | Supports ANSI/ISO standard C with Analog Devices extensions. Default mode. For more information, see “-analog” on page 1-20. |
| -c++ | Supports ANSI/ISO standard C++ with Analog Devices extensions. For more information, see “-c++” on page 1-20. |
| -traditional | Supports pre-ANSI K&R C. For more information, see “-traditional” on page 1-20. |

Table 1-3. C/C++ Compiler Common Switches

| Switch Name | Description |
|---------------------|---|
| sourcefile | Specifies the file to be compiled. For more information, see “sourcefile” on page 1-21. |
| -@ filename | Reads command-line input from the file. For more information, see “-@ filename” on page 1-21. |
| -21532 or -21535 | Generates code for ADSP-2153x DSPs. For more information, see “-21352, -21535” on page 1-21. |
| -A symbol[tokens] | Asserts the specified name as a predicate. For more information, see “-A name[tokens]” on page 1-21. |
| -alttok | Allows alternative keywords and sequences in sources. For more information, see “-alttok” on page 1-22. |
| -auto-inline factor | Controls how much the compiler automatically inlines functions. For more information, see “-auto-inline factor” on page 1-22. |
| -BLACKFIN | Generates code for Blackfin DSPs. For more information, see “-BLACKFIN” on page 1-23. |
| -build-lib | Directs the librarian to build a library file. For more information, see “-build-lib” on page 1-23. |
| -C | Retains preprocessor comments in the output file; active only with the -E or -P switch. For more information, see “-C” on page 1-23. |
| -c | Compiles and/or assembles only, but does not link. For more information, see “-c” on page 1-24. |
| -circbuf | Causes the compiler to treat array references as circular buffer references, and to generate code accordingly. For more information, see “-circbuf” on page 1-24. |
| -const-read-write | Specifies that data accessed via a pointer to const data may be modified elsewhere. For more information, see “-const-read-write” on page 1-24. |
| -debug-types | Supports building a *.h file directly and writing a complete set of debugging information for the header file. For more information, see “-debug-types” on page 1-24. |

Compiler Command-Line Interface

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

| Switch Name | Description |
|------------------------------|--|
| <code>-Dmacro[=def]</code> | Defines <code>macro</code> . For more information, see “-Dmacro[=definition]” on page 1-25. |
| <code>-dry</code> | Displays, but does not perform, main driver actions (verbose dry run). For more information, see “-dry” on page 1-25. |
| <code>-dryrun</code> | Displays, but does not perform, top-level driver actions (terse dry run). For more information, see “-dryrun” on page 1-25. |
| <code>-E</code> | Preprocesses, but does not compile, the source file. For more information, see “-E” on page 1-25. |
| <code>-EE</code> | Preprocesses and compiles the source file. For more information, see “-EE” on page 1-26. |
| <code>-extra-keywords</code> | Recognizes Blackfin DSP extensions to ANSI/ISO standards for C. Default mode. For more information, see “-extra-keywords” on page 1-26. |
| <code>-flags-tool</code> | Passes command-line switches through the compiler to other build tools. For more information, see “-flags{-asm -compiler -lib -link} switch [,switch2 [,...]]” on page 1-26. |
| <code>-full-version</code> | Displays the version number of the driver and processes invoked by the driver. For more information, see “-full-version” on page 1-26. |
| <code>-g</code> | Generates DWARF-2 debug information. For more information, see “-g” on page 1-26. |
| <code>-H</code> | Outputs a list of included header files, but does not compile. For more information, see “-H” on page 1-27. |
| <code>-HH</code> | Outputs a list of included header files and compiles. For more information, see “-HH” on page 1-27. |
| <code>-h[elp]</code> | Outputs a list of command-line switches with brief syntax descriptions. For more information, see “-h[elp]” on page 1-27. |
| <code>-I directory</code> | Appends <code>directory</code> to the standard search path. For more information, see “-I directory [{, ;} directory...]” on page 1-28. |

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

| Switch Name | Description |
|---------------------------------|---|
| <code>-include filename</code> | Includes named file prior to each source file. For more information, see “-include filename” on page 1-28. |
| <code>-ipa</code> | Specifies that interprocedural analysis should be performed for optimization between translation units. For more information, see “-ipa” on page 1-28. |
| <code>-jcs2l</code> | Enables the conversion of short jumps to long jumps when necessary. For more information, see “-jcs2l” on page 1-28. |
| <code>-L directory</code> | Appends <i>directory</i> to the standard library search path. For more information, see “-L directory[{,;} directory...]” on page 1-28. |
| <code>-l library</code> | Searches <i>library</i> for functions when linking. For more information, see “-traditional” on page 1-20. |
| <code>-M</code> | Generates make rules only, but does not compile. For more information, see “-M” on page 1-29. |
| <code>-MM</code> | Generates make rules and compiles. For more information, see “-MM” on page 1-29. |
| <code>-Mt filename</code> | Makes dependencies for the specified source file. For more information, see “-Mt filename” on page 1-29. |
| <code>-map filename</code> | Directs the linker to generate a memory map of all symbols. For more information, see “-map filename” on page 1-30. |
| <code>-no-alttok</code> | Does not allow alternative keywords and sequences in sources. For more information, see “-no-alttok” on page 1-30. |
| <code>-no-builtin</code> | Disable recognition of <code>__builtin</code> functions. For more information, see “-no-builtin” on page 1-30. |
| <code>-no-defs</code> | Disables preprocessor definitions: macros, include directories, library directories, run-time headers, or keyword extensions. For more information, see “-no-defs” on page 1-30. |
| <code>-no-extra-keywords</code> | Does not define language extension keywords which could be valid C/C++ identifiers. For more information, see “-no-extra-keywords” on page 1-30. |

Compiler Command-Line Interface

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

| Switch Name | Description |
|------------------|---|
| -no-inline | Ignores the <code>inline</code> keyword. For more information, see “-no-inline” on page 1-31. |
| -no-int-to-fract | Prevents the compiler from turning integer into fractional arithmetic. For more information, see “-no-int-to-fract” on page 1-31. |
| -no-jcs2l | Disables the conversion of short jumps to long jumps. For more information, see “-no-jcs2l” on page 1-31. |
| -no-restrict | Disables the <code>restrict</code> keyword. For more information, see “-no-restrict” on page 1-31. |
| -no-std-def | Disables normal macro definitions and also ADI keyword extensions that do not have leading underscores (<code>_</code>). For more information, see “-no-std-def” on page 1-31. |
| -no-std-inc | Searches only for preprocessor include header files in the current directory and in directories specified with the <code>-I</code> switch. For more information, see “-no-std-inc” on page 1-32. |
| -no-std-lib | When linking, searches for only those library files specified with the <code>-l</code> switch. For more information, see “-no-std-lib” on page 1-32. |
| -nothreads | Specifies that no support is required for multi-threaded applications. For more information, see “-nothreads” on page 1-32. |
| -O | Enables code optimizations. For more information, see “-O” on page 1-32. |
| -Ofp | Offsets the Frame Pointer to allow more short load and store instructions. Not allowed with <code>-g</code> . For more information, see “-Ofp” on page 1-33. |
| -Os | Optimizes the file to decrease code size. For more information, see “-Os” on page 1-33. |
| -Ox | Aggressively removes sign-extensions to 32-bits, for DSP code that uses <code>short ints</code> for expressions. Inappropriate for generic C code. For more information, see “-Ox” on page 1-33. |

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

| Switch Name | Description |
|--------------------------------|--|
| -Oz | Specifies that program data size is less than 64K, so high order half of address is zero. For more information, see “-Oz” on page 1-34. |
| -o <i>filename</i> | Specifies the output file name. For more information, see “-o filename” on page 1-34. |
| -P | Preprocesses, but does not compile, the source file. Output does not contain <code>#line</code> directives. For more information, see “-P” on page 1-34. |
| -PP | Preprocesses and compiles the source file. Output does not contain <code>#line</code> directives. For more information, see “-PP” on page 1-34. |
| -p[1 2] | Generates profiling instrumentation. For more information, see “-p[1 2]” on page 1-35. |
| -path-tool <i>directory</i> | Uses the specified directory as the location of the specified compilation tool (assembler, compiler, library builder, or linker). For more information, see “-path [-asm -compiler -lib -link] directory” on page 1-35. |
| -path-install <i>directory</i> | Uses the specified directory as the location of all compilation tools. For more information, see “-path-install directory” on page 1-35. |
| -path-output <i>directory</i> | Specifies the location of non-temporary files. For more information, see “-path-output directory” on page 1-35. |
| -path-temp <i>directory</i> | Specifies the location of temporary files. For more information, see “-path-temp directory” on page 1-36. |
| -pedantic | Issues compiler warnings for constructs that are not strictly ISO/ANSI standard C/C++ compliant. For more information, see “-pedantic” on page 1-36. |
| -pedantic-errors | Issues compiler errors for constructs that are not strictly ISO/ANSI standard C/C++ compliant. For more information, see “-traditional” on page 1-20. |
| -pplist <i>filename</i> | Outputs a raw preprocessed listing to the specified file. For more information, see “-pplist filename” on page 1-36. |

Compiler Command-Line Interface

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

| Switch Name | Description |
|---------------------------------------|--|
| <code>-proc <i>processorID</i></code> | Specifies a processor for which the compiler should produce suitable code. For more information, see “-proc processorID” on page 1-37. |
| <code>-R <i>directory</i></code> | Appends <i>directory</i> to the standard search path for source files. For more information, see “-R directory[{: ,}directory ...]” on page 1-37. |
| <code>-restrict</code> | Enables the <code>restrict</code> keyword. For more information, see “-restrict” on page 1-37. |
| <code>-S</code> | Stops compilation before running the assembler. For more information, see “-S” on page 1-38. |
| <code>-s</code> | When linking, removes debugging information from the output executable file. For more information, see “-s” on page 1-38. |
| <code>-save-temps</code> | Saves intermediate files. For more information, see “-save-temps” on page 1-38. |
| <code>-sat32</code> | Saturates all accumulations at 32 bits, which is the default. For more information, see “-sat32” on page 1-38. |
| <code>-sat40</code> | Saturates all accumulations at 40 bits rather than the default 32 bits. For more information, see “-sat40” on page 1-38. |
| <code>-show</code> | Displays the driver command-line information. For more information, see “-show” on page 1-39. |
| <code>-signed-char</code> | Makes the default type for <code>char</code> signed. For more information, see “-signed-char” on page 1-39. |
| <code>-syntax-only</code> | Checks the source code for compiler syntax errors, but does not write any output. For more information, see “-syntax-only” on page 1-39. |
| <code>-T <i>filename</i></code> | Specifies the Linker Description File. For more information, see “-T filename” on page 1-39. |
| <code>-threads</code> | Enables the support for multithreaded applications. For more information, see “-threads” on page 1-39. |

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

| Switch Name | Description |
|------------------------------|---|
| -time | Displays the elapsed time as part of the output information on each part of the compilation process. For more information, see “-time” on page 1-40. |
| -traditional | Applies traditional C compiler rules (consistent with pre-ANSI K&R C compilers). For more information, see “-traditional” on page 1-20. |
| -Umacro | Undefines <i>macro</i> . For more information, see “-Umacro” on page 1-40. |
| -unsigned-char | Makes the default type for <code>char</code> unsigned. For more information, see “-unsigned-char” on page 1-40. |
| -v | Displays version and command-line information for all compilation tools. For more information, see “-v” on page 1-40. |
| -verbose | Displays command-line information for all compilation tools. For more information, see “-verbose” on page 1-40. |
| -version | Displays version information for all compilation tools. For more information, see “-version” on page 1-40. |
| -warn-protos | Issues warnings about functions without prototypes. For more information, see “-warn-protos” on page 1-41. |
| -Werror <i>number</i> | Overrides the default severity of the specified error message. For more information, see “-Werror-limit number” on page 1-41. |
| -Wdriver-limit <i>number</i> | Halts the driver after reaching the specified number of errors. For more information, see “-Wdriver-limit number” on page 1-41. |
| -Werror-limit <i>number</i> | Stops compiling after reaching the specified number of errors. For more information, see “-Werror-limit number” on page 1-41. |
| -Wremarks | Issues compiler remarks. For more information, see “-Wremarks” on page 1-41. |
| -Wterse | Issues the briefest form of compiler warning, errors, and remarks. For more information, see “-Wterse” on page 1-42. |

Compiler Command-Line Interface

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

| Switch Name | Description |
|-----------------------------|--|
| <code>-w</code> | Disables all warnings. For more information, see “-w” on page 1-42. |
| <code>-write-files</code> | Enables compiler I/O redirection. For more information, see “-write-files” on page 1-42. |
| <code>-xref filename</code> | Outputs cross-reference information to the specified file. For more information, see “-xref filename” on page 1-42. |

Table 1-4. C++ Mode Compiler Switches

| Switch Name | Description |
|---------------------------------|--|
| <code>-explicit</code> | Supports the <code>explicit</code> specifier on constructor declarations. This is the default mode. For more information, see “-explicit” on page 1-43. |
| <code>-instant[all used]</code> | Instantiates all or used members of a class. For more information, see “-instant[all used]” on page 1-43. |
| <code>-namespace</code> | Supports namespaces. This is the default mode. For more information, see “-namespace” on page 1-44. |
| <code>-newforinit</code> | Limits the scope of any symbol declared within a “for” statement. For more information, see “-newforinit” on page 1-44. |
| <code>-newvec</code> | Allows the overloading of <code>new[]</code> and <code>delete[]</code> . For more information, see “-newvec” on page 1-44. |
| <code>-no-demangle</code> | Prevents filtering of any linker errors through the demangler. For more information, see “-no-demangle” on page 1-44. |
| <code>-no-explicit</code> | Does not support the <code>explicit</code> specifier on constructor declarations. For more information, see “-no-explicit” on page 1-44. |
| <code>-no-namespace</code> | Does not support namespaces. For more information, see “-no-namespace” on page 1-44. |
| <code>-no-newvec</code> | Does not allow the overloading of <code>new[]</code> and <code>delete[]</code> . For more information, see “-no-newvec” on page 1-45. |
| <code>-notstrict</code> | Omits warning and/or error messages for non-ANSI constructs. For more information, see “-notstrict” on page 1-45. |

Table 1-4. C++ Mode Compiler Switches (Cont'd)

| Switch Name | Description |
|-------------|--|
| -no-wchar | Disables <code>wchar_t</code> keyword. For more information, see “-no-wchar” on page 1-45. |
| -strict | Generates error messages for non-ANSI constructs. For more information, see “-strict” on page 1-45. |
| -strictwarn | Generates warning messages for non-ANSI constructs. For more information, see “-strictwarn” on page 1-45. |
| -tpautooff | Disables automatic instantiation of templates. For more information, see “-tpautooff” on page 1-46. |
| -trdforinit | Limits the scope of any symbol declared within a “for” statement. For more information, see “-trdforinit” on page 1-46. |
| -typename | Recognizes the <code>typename</code> keyword. This is the default mode. For more information, see “-typename” on page 1-46. |
| -wchar | Enables new <code>wchar_t</code> . For more information, see “-wchar” on page 1-46. |

C/C++ Mode Selection Switch Descriptions

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

-analog

The `-analog` (Analog Devices C compilation) switch directs the compiler to support Analog Devices extensions to ANSI/ISO standard C language. This is the default mode. For more information about these extensions, see [“C/C++ Compiler Language Extensions” on page 1-55](#).

-c++

The `-c++` (C++ mode) switch directs the compiler to assume that the source file(s) are written in ANSI/ISO standard C++ language with Analog Devices language extensions. When using this switch, source files with an extension of `.c` will be compiled in C++ mode.

-traditional

The `-traditional` (traditional compilation) switch directs the compiler to apply the following rules (consistent with pre-ANSI K&R C compilers) to compilation.

- All `extern` declarations (including implicit declarations of functions) take effect globally.
- Analog Devices C/C++ language extensions are disabled except for the forms of the extra keywords that begin with a double underscore (`__`).
- Pointer/integer comparisons are always allowed.

C/C++ Compiler Common Switch Descriptions

The following command-line switches apply in 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. Refer to [Table 1-1 on page 1-8](#) for more information.

-@ filename

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

-21352, -21535

The `-21532` or `-21535` (compile for ADSP-2153x) switch directs the build tools chain to generate code suitable for the corresponding ADSP-2153x DSP. When compiling with the switch, the `__ADSP21532__`, `__ADSP21535__`, or `__ADSPBLACKFIN__` preprocessor macros are defined as 1.

-A name[tokens]

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

```
system()    machine()    cpu()    compiler()
embedded    adspblkfn    ccblkfn
```

Compiler Command-Line Interface

-alttok

The `-alttok` (alternative token) switch directs the compiler to allow alternative operator keywords and digraph sequences in source files. This is the default mode.

ANSI C trigraphs sequences are always expanded (even with the `-no-alttok` option), and digraph sequences are expanded only in C source files.

The following operator keywords are enabled by default.

| Keyword | Equivalent |
|---|-------------------------|
| <code>and</code> | <code>&&</code> |
| <code>and_eq</code> | <code>&=</code> |
| <code>bitand</code> | <code>&</code> |
| <code>bitor</code> | <code> </code> |
| <code>compl</code> | <code>~</code> |
| <code>or</code> | <code> </code> |
| <code>or_eq</code> | <code> =</code> |
| <code>not</code> | <code>!</code> |
| <code>not_eq</code> | <code>!=</code> |
| <code>xor</code> | <code>^=</code> |
| To use these keywords in C, you should use <code>#include <iso646.h></code> . | |

-auto-inline factor

The `-auto-inline` (auto inline) switch directs the optimizer to automatically inline functions where the reduction in execution time justifies the increase in code size. The amount of effected inlining is specified by *factor*,

which is a floating-point number that determines how aggressively functions are inlined. The amount of inlining effected is shown by the following examples.

| | |
|--------|--|
| 0.0 | Reject all inlining |
| 1.0 | Inline if code size increase does not exceed speed improvement |
| 10.0 | Inline allowing a reasonable amount of code size increase |
| 1000.0 | Inline practically everything |

-BLACKFIN

The `-BLACKFIN` (compile for Blackfin DSP) switch directs the build tools chain to generate code suitable for the generic Blackfin DSPs. When compiling with this switch, the `__ADSPBLACKFIN__` preprocessor macro is defined as 1.



The `-BLACKFIN` switch supports only the basic DSP architecture. It does not support peripherals.

-build-lib

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

-C

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

Compiler Command-Line Interface

-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 (`.obj`) for each source file.

-circbuf

The `-circbuf` (circular buffer) switch causes the compiler to treat array references, such as `"array_var[i%n]"`, as circular buffer references, and to generate code accordingly, if possible.

-const-read-write

The compiler's default behavior assumes that data referenced through `const` pointers will never change. 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.

-debug-types

The `-debug-types` option provides for building an `*.h` file directly and writing a complete set of debugging information for the header file. The `-g` option 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`.

-Dmacro[=*definition*]

The `-D` (define macro) switch directs the compiler to define a macro. If you do not include the optional definition string, the compiler defines the macro as the string `'1'`. If *definition* is required to be a character string constant then it must be surrounded by escaped double quotes. Note that the compiler processes `-D` switches on the command line before any `-U` (undefine macro) switches.



This switch can be invoked with the **Definitions:** dialog field located in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Preprocessor** category.

-dry

The `-dry` (a verbose dry run) switch directs the compiler to display main driver actions, but not to perform them. This will also cause the compiler to display any options it passes on and then perform a dry run.

-dryrun

The `-dryrun` (a terse dry run) switch directs the compiler to display top-level driver 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 `<standout>` unless the output file is specified with the `-o` switch. Note that the `-C` switch can be used in combination with the `-E` switch.



You can invoke it with the **Stop after: Preprocessor** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

Compiler Command-Line Interface

-EE

The `-EE` (run after preprocessing) switch is similar to the `-E` switch, but it does not halt compilation after preprocessing.

-extra-keywords

The `-extra-keywords` (enable short-form keywords) switch directs the compiler to recognize the ADI 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. The supported keywords are `asm`, `inline`, `restrict`, `section`, `bool`, `false`, and `true`.

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

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

Table 1-5. Build Tools' Options

| Option | Tool |
|------------------------------|-----------------|
| <code>-flags-asm</code> | Assembler |
| <code>-flags-compiler</code> | Compiler |
| <code>-flags-lib</code> | Library Builder |
| <code>-flags-link</code> | Linker |


-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.

-g

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

When `-g` is used without `-O`, the `-no-inline` option is implied. If the `-g` switch is used in conjunction 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 through VisualDSP++. This combination of options provides line debugging and global variable debugging.

 When `-g` and `-O` are specified, no debug information is available for local variables and the standard optimizations can sometimes re-arrange program code in a way that inaccurate line number information may be produced. For full debugging capabilities, use the `-g` switch without the `-O` switch.

You can invoke this switch by selecting the **Generate debug information** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-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 may be used to specify the redirection of the list to a file.

-HH

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

-h[elp]

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

Compiler Command-Line Interface

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

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

-include *filename*

The **-include** (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

This switch turns on Interprocedural Analysis (IPA) in the compiler. This enables optimization across the entire program, including source files that were compiled separately. For best results, **-ipa** should be applied to all files in the program. Specifying **-ipa** also implies **-O**. [For more information, see “Interprocedural Analysis” on page 1-52.](#)

-jcs2l

This switch requests the linker to convert compiler-generated short jumps to long jumps when necessary. This is the default mode.

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

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

-l *library*

The **-l** (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 `.dlib` 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`.

Normally, list all object files on the command line before using the `-l` switch; this ensures that functions referred to by object files are loaded from the library in the given order. This switch may be specified more than once; libraries are searched as encountered during the left-to-right processing of the command line.

-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 ...
```

-MM

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

-Mt filename

The `-Mt filename` (output make rule for the named source) switch specifies the name of the source file for which the compiler generates the make rule when you use the `-M` or `-MM` switch. If the named file is not in the current directory, you must provide the path name in double quotation marks (`"`). The new file name will override the default `base.doj`. The `-Mt` option supports the `.IMPORT` extension.

Compiler Command-Line Interface

-map *filename*

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

-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. For more information, see [“-alttok” on page 1-22](#).

-no-builtin

The `-no-builtin` (no built-in functions) switch directs the compiler to ignore any built-in functions that do not begin with two underscores (`__`). Note that this switch influences many functions. This switch also predefines the `__NO_BUILTIN` preprocessor macro. For more information on built-in functions, see [“Built-In Functions” on page 1-76](#).

-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. It also disables the Analog Devices C/C++ keyword extensions.

-no-extra-keywords

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

-no-inline

The `-no-inline` (disable inline keyword) switch directs the compiler not to perform any high-level optimizations associated with function inlining.

-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 a statement such as

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

may be changed, by default, into a fractional multiplication. The saturation properties of integer and fractional arithmetic are different; therefore, if the expression overflows, then the results will differ. Specifying the `-no-int-to-fract` switch disables this optimization.

-no-jcs2l

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

-no-restrict

The `-no-restrict` (disable restrictions) switch directs the compiler to disable recognition of the `restrict` keyword as a type qualifier for pointers and array parameter to functions.


-no-std-def

The `-no-std-def` (disable standard macro definitions) switch prevents the compiler from defining default preprocessor macro definitions. Note that this switch also disables the ADI keyword extensions that have no leading underscores, such as `asm`.

Compiler Command-Line Interface

-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.

 You can invoke this switch by selecting the **Ignore standard include paths** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Preprocessor** category.

-no-std-lib


The `-no-std-lib` (disable standard library search) switch directs the linker to limit its search to those libraries specified with the `-l` switch.

-nothreads

The `-nothreads` (disable thread-safe build) switch specifies that all compiled code and libraries used in the build need not be thread safe. This is the default setting.

-O

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.

 You can invoke this switch by selecting the **Enable optimization** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-Ofp

The `-Ofp` (frame pointer optimization) switch directs the compiler to offset the Frame Pointer within a function, if this will allow the compiler to use more short load and store instructions. This switch may not be used with the `-g` debugging switch, since the debugger would not be able to find the local procedural information. Specifying `-Ofp` also implies `-O`.

-Os

The `-Os` (enable code size optimization) switch directs the compiler to perform standard optimizations with an emphasis on reducing code size rather than on improving code performance.

-Ox

The `-Ox` (sign extension optimization) switch directs the compiler to aggressively remove sign extensions from `short int` to `int` during expression evaluation. This switch is intended for DSP algorithms that perform computations using 16-bit input values. These algorithms will not produce final or intermediate values that are outside the range of values expressed by 16-bit types.

The `-Ox` switch allows the compiler to “cut back” the intermediate results without affecting the validity of the program. This switch can be used because Blackfin DSPs provide efficient 16-bit arithmetic. Specifying `-Ox` also implies the `-O` switch.



This switch should not be used unless the code has been specifically written with such truncation in mind.

Here is an example that fails with `-Ox`:

```
short a = 32000, b = 32000;

if (a + b < 0) ...
```

Compiler Command-Line Interface

The condition will be true with `-0x` as the 16-bit addition overflows, producing a negative result. Without `-0x`, the standard interpretation uses `int` (32-bit) arithmetic to yield a sum of +64000.

-Oz

The `-Oz` (low program memory) switch informs the compiler that program data has been placed in the lowest 64K area of memory. This allows the compiler to load data addresses with a single instruction, instead of two, thereby reducing code size and increasing performance. Specifying `-Oz` also implies the `-O` switch.



You must arrange for this memory layout by using a customized `.LDF` file.

--o *filename*

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

-P

The `-P` (omit line numbers) switch directs the compiler to stop after the C/C++ preprocessor runs (without compiling) and to omit the `#line` preprocessor directives (with line number information) in the output from the preprocessor. The `-C` switch can be used in conjunction with `-P` 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` (generate profiling implementation) switch directs the compiler to generate the additional instructions needed to profile the program by recording the number of cycles spent in each function.

The `-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 both `mon.out` and the standard output stream. For more information on profiling, see [“Profiling with Instrumented Code” on page 1-95](#).

-path [-asm | -compiler | -lib | -link] *directory*

The `-path` (tool location) switch directs the compiler to use the specified directory as the location of the specified compilation tool. Respectively, the tools are the assembler, compiler, librarian, and linker. Use this switch when you wish to override the normal version of one or more of the tools. The `-path-tool` option overrides the directory specified by the `-path-install` switch.

-path-install *directory*

The `-path-install` (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-tool` switch.

-path-output *directory*

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

Compiler Command-Line Interface

-path-temp *directory*

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

-pedantic

The `-pedantic` (ANSI standard warning) switch causes the compiler to issue a warning for each construct found in your program that does not strictly conform to ANSI/ISO standard C or C++ language. Note that the compiler may not detect all such constructs. In particular, the `-pedantic` switch does not cause the compiler to issue errors when ADI keyword extensions are used.

-pedantic-errors

The `-pedantic-errors` (ANSI standard errors) switch causes the compiler to issue an error instead of a warning for cases described in the `-pedantic` switch.

-pplist filename

The `-pplist` (preprocessor listing) directs the preprocessor to output a listing to the named file. When more than one source file has been 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. Each listing line begins with a key character that identifies its type.

| Character | Meaning |
|-----------|---|
| N | Normal line of source |
| X | Expanded line of source |
| S | Line of source skipped by <code>#if</code> or <code>#ifdef</code> |
| L | Change in source position |

| Character | Meaning |
|-----------|---|
| R | Diagnostic message (remark) |
| W | Diagnostic message (warning) |
| E | Diagnostic message (error) |
| C | Diagnostic message (catastrophic error) |

-proc processorID

The `-proc` (target processor) switch specifies that the compiler should produce code suitable for the specified DSP. If the processor identifier is unknown to the compiler, it attempts to read required switches for code generation from the file `<identifier>.ini`. The compiler searches for this file in the VisualDSP++ System folder.

-R directory[{:|,}*directory* ...]

The `-R` (add source directory) switch directs the compiler to add the specified directory to the list of directories searched for source files.

On Windows platforms, multiple source directories are given as a colon-, comma-, or semicolon-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) and contain drive specifiers (on Windows platforms) are not affected by this option.

-restrict

The `-restrict` (restriction) switch directs the compiler to recognize the `restrict` keyword as a type qualifier for pointers and function parameter arrays that decay to pointers. This is the default setting.

Compiler Command-Line Interface

-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.



You can invoke this switch by selecting the **Stop after: Compiler** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-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.

-sat32

The `-sat32` (32-bit saturation) switch directs the compiler to saturate at 32 bits. This is the default setting.

-sat40

The `-sat40` (40-bit saturation) switch directs the compiler to saturate at 40 bits, rather than at the default which saturates at 32 bits.

-save-temps

The `-save-temps` (save intermediate files) switch directs the compiler not to discard intermediate files. The compiler places the intermediate [temporary] output (`*.i`, `*.is`, `*.s`, `*.doj`) files in the current directory. See [Table 1-1 on page 1-8](#) for a list of intermediate files.

The location of the saved file is affected by the `-path-output` switch, if provided. That switch sets the path for all “permanent” outputs that do not otherwise have a path set, the object file included.

-show

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

-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.

-syntax-only

The `-syntax-only` (just check syntax) switch directs the compiler to check the source code for syntax errors, but not write any output.

-T *filename*

The `-T` (Linker Description File) switch directs the linker to use the specified Linker Description File as control input for linking. If `-T` is not specified, a default `.LDF` is selected based on the processor used.

-threads

The `-threads` (enable thread-safe build) switch specifies that the build and link are thread-safe. The macro `__ADI__THREADS` is defined to one (1). It is used for conditional compilation by the preprocessor and by default `.LDF` files to link with thread-safe libraries.



This switch is likely to be used only by applications involving the VisualDSP++ Kernel (VDK).

Compiler Command-Line Interface

-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.

-Umacro

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

-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.

-warn-protos

The `-warn-protos` (prototype warning) switch directs the compiler to produce a warning message when a function is called without a full prototype being supplied.

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

The `-W` (override error message) switch directs the compiler to override the severity of the specified diagnostic messages (errors, remarks, or warnings). The *number* argument specifies the message to override.

At compilation time, the compiler produces a number for each specific compiler diagnostic message. The `-D` (discretionary) suffix attached to the message number marks a message whose severity can be overridden. Each diagnostic message is identified by a number that is used across all compiler software releases.

-Wdriver-limit *number*

The `-Wdriver-limit` (maximum process errors) switch sets a maximum number of driver errors (command line, etc.) at which the driver aborts.

-Werror-limit *number*

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

-Wremarks

The `-Wremarks` (enable diagnostic warnings) switch directs the compiler to issue remarks, which are diagnostic messages that are even milder than warnings.



You can invoke this switch by selecting the **Enable remarks** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Warning** selection.

Compiler Command-Line Interface

-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.



You can invoke this switch by selecting the **Disable all warnings and remarks** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Warning** selection.

-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.

-xref filename

The `-xref` (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 form

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

is written to the named file. The `symbol-id` represents a unique decimal number for the symbol, and `ref-code` is one of the following 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) |

C++ Mode Compiler Switch Descriptions

The following switches apply only to C++ compiler.

-explicit

The `-explicit` (explicit specifier) switch directs the compiler to enable support for the `explicit` specifier on constructor declarations. The compiler defines the `__EXPLICIT` preprocessor macro. This option is enabled by default.

-instant[all|used]

By default, the compiler suppresses the instantiation of any templates on the first compilation. It also allows the prelinker to determine which files need to be recompiled to instantiate the required templates. However, the `-instantused` switch instantiates any template entities that are used in the first compilation, and the `-instantall` switch instantiates all template entities regardless of whether they are used or not. Both of these options can be used in combination with the prelinker.

Compiler Command-Line Interface

-namespace

The `-namespace` (namespace) switch directs the compiler to enable support for namespaces. This is the default mode.

-newforinit

The `-newforinit` (new for initialization) switch directs the compiler to limit a scope of any declaration within a `for` statement to the block contained within that `for` statement.

-newvec

The `-newvec` (new vector) switch directs the compiler to allow the overloading of the `new[]` and `delete[]` operators. The compiler also defines the `__ARRAY_OPERATORS` macro when using this option or any another option that enables overloading of the dynamic memory allocation operators. This is the default mode.

-no-demangle

The `-no-demangle` (disable demangler) switch directs the compiler to prevent the driver from filtering linker errors through the demangler. The demangler's primary role is to convert the encoded name of a function into a more understandable version of the name.

-no-explicit

The `-no-explicit` (disable explicit specifier) switch directs the compiler to disable support for the explicit specifier on constructor declarations. For more information, see [“-explicit” on page 1-43](#).

-no-namespace

The `-no-namespace` (disable namespace) switch directs the compiler to disable support for namespaces.

-no-newvec

The `-no-newvec` (disallow a new vector) switch directs the compiler to disallow the overloading of the `new[]` and `delete[]` operators. For more information, see [“-newvec” on page 1-44](#).

-notstrict

The `-notstrict` (non-strict compilation) switch directs the compiler to omit diagnostic messages (warnings and errors) for any constructs in a C++ source file that do not conform to the ANSI standard for the C++ programming language.

-no-wchar

The `-no-wchar` (disable wide char type) switch directs the compiler to disable the `wchar_t` keyword.

-strict

The `-strict` (strict standard) switch directs the compiler to generate diagnostic error messages for any constructs of a source file that do not conform to the ANSI standard for the C++ programming language. The `-strict` switch defines the `__STRICT_ANSI__` macro.

-strictwarn

The `-strictwarn` (warn if non-strict) switch directs the compiler to generate diagnostic warning messages for any constructs of a source file that do not conform to the ANSI standard for the C++ programming language. The `-strictwarn` switch defines the `__STRICT_ANSI__` macro.

Compiler Command-Line Interface

-tpautooff

The `-tpautooff` (disable automatic template instantiation) switch directs the compiler to disable automatic instantiation of templates. It also prevents implicit inclusion of source files as a method of finding definitions of template entities to be instantiated.

-trdforinit

The `-trdforinit` (traditional initialization) switch directs the compiler to limit a scope of any declaration within a `for` statement to the block containing that `for` statement.

-typename

The `-typename` (type name) switch directs the compiler to recognize the `typename` keyword and to define the `__TYPENAME` macro. This is the default mode.

-wchar

The `-wchar` (enable wide char type) switch directs the compiler to enable the `wchar_t` keyword.

Data Type Sizes

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. [Table 1-6](#) shows the sizes used by each of the intrinsic C/C++ data types.

Table 1-6. Data Type Sizes for ADSP-2153x DSPs

| Type | Bit Size | sizeof returns |
|------------------|--------------------|----------------|
| int | 32 bits signed | 4 |
| unsigned int | 32 bits unsigned | 4 |
| long | 32 bits signed | 4 |
| unsigned long | 32 bits unsigned | 4 |
| char | 8 bits signed | 1 |
| unsigned char | 8 bits unsigned | 1 |
| short | 16 bits signed | 2 |
| unsigned short | 16 bits unsigned | 2 |
| pointer | 32 bits | 4 |
| function pointer | 32 bits | 4 |
| float | 32 bits float | 4 |
| double | 32 bits float | 4 |
| fract16 | 16 bits fractional | 2 |
| fract32 | 32 bits fractional | 4 |

In the Blackfin DSP architecture, the `long long int`, `unsigned long long int`, and `long double` data types are not implemented (they will not be redefined to other types). In general, floating data types should be expected to run more slowly, relying largely on software-emulated arithmetic.

Compiler Command-Line Interface

Type `double` can pose a problem. The C language tends to default to `double` for constants and floating-point calculations. Without some special handling, many programs inadvertently use slow-speed emulated 64-bit floating-point arithmetic, even when variables are declared consistently as `float`. To avoid this problem and provide the best performance, the size of `double` on Blackfin DSPs is 32 bits. This should be acceptable for most DSP programming. However, it does not conform fully to the ANSI C standard.

Standard `#include` files automatically redefine the math library interfaces, allowing functions such as `sin` to be directly called with the proper size operands. Therefore:

```
float sinf (float);    /* 32-bit */  
double sin (double);  /* 32-bit */
```

For descriptions of these functions and their implementation, see Chapter 2, [“C Run-Time Library Reference” on page 2-23](#).

Optimization Control

The compiler can operate at several levels of optimization. The following list identifies the levels with least optimization listed first and most optimization listed last.

- **Debugging** — The compiler produces debug information to ensure that the object code matches the appropriate source code line. See [“-g” on page 1-26](#) for more information.
- **Default** — The compiler performs basic high-level optimization, such as inlining functions that are explicitly marked for inlining.

- **Procedural optimization** — The compiler performs advanced, aggressive optimization on each procedure in the file being compiled. If debugging is also requested, the optimization is given priority so the debugging functionality may be limited. See “-O” on page 1-32 for more information.
- **Interprocedural optimization** — The compiler performs advanced, aggressive optimization over the whole program, in addition to the per-file optimizations in procedural optimization. See “-ipa” on page 1-28 for more information.

The `ccblkn` compiler offers several switches (and VisualDSP++ options) to control the level and type of optimizations that are applied to C and C++ source.

Interprocedural analysis (see “Interprocedural Analysis” on page 1-52) allows the compiler to see all of the source files that are used and to use that information to enable the other optimizations to be exploited as fully as possible.

When no optimization switches are specified, the compiler effects only basic high-level optimizations, such as inlining functions, which have been explicitly marked for inlining. When `-g` is specified, all inlining is suppressed to provide comprehensive debugging information. When `-inline` is specified with `-g`, the explicitly-specified inlining is provided, reducing the amount of source line debug information that is available. Therefore, using `-g` by itself effectively disables most optimizations.

Normally, a program is optimized to process the data as quickly as possible, but in some circumstances, the program speed is less important than reducing the size of the generated code. When the reduced code size is more important, use the `-Os` switch to direct the compiler to perform only standard optimizations. (See “-O” on page 1-32 for more information.)

The `-O` switch requests the compiler to affect all generally safe optimizations. It also requests the compiler to generate the fastest possible executing code while conforming to standard language interpretations and

a conservative view of any possible interactions between variables. The interprocedural analysis enables the compiler to be more aggressive in optimizing the program since it has more information of the overall structure of the program and the data being manipulated by the program.

Several options exist that notify the compiler about certain assumptions how that data may be being processed for better code optimization. If the assumptions are not true, then the program's behavior is undefined. These options are:

- `-Ofp` — Tells 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 not be able to interpret the resulting frame structure; therefore, this option is not allowed in conjunction with `-g`. See “[-Ofp](#)” on page 1-33 for more information.
- `-Ox` — Directs the compiler to aggressively remove sign extensions from `short int` to `int` during expression evaluation. This switch is intended for DSP algorithms where the values resulting from expressions containing 16-bit values are expected to be valid 16-bit values. The `-Ox` switch allows the compiler to “cut back” the intermediate results without affecting the validity of the program. This switch can be used because Blackfin DSPs provide efficient 16-bit arithmetic.



This switch should not be used unless the code has been specifically written with such truncation in mind. See “[-Ox](#)” on page 1-33 for more information.

- `-Oz` — Tells the compiler that the program’s data is in the low 64K of memory, so only the low 16 bits of address registers need to be loaded, and the high-order address bits will always be zero.



This switch does not place data in low memory — it tells the compiler that the data has been placed in low memory. You do the actual positioning of the data via the `.LDF` file. See “`-Oz`” on [page 1-34](#) for more information.

The use of the `-Ofp`, `-Ox`, or `-Oz` optimization switches also implies the use of the `-O` switch since they enable more aggressive optimizations to be applied to the program.

The optimizer attempts to vectorize loops when it is safe to do so and uses information from the Interprocedural Analyzer to identify more opportunities to do so. In addition, there may be other loops that you know are safe candidates for the vectorizer; you can use pragmas to inform the optimizer of such loops (see “[Pragmas to Assist in Vectorizing Loops](#)” on [page 1-90](#)).

Inlining Control

By default, the compiler inlines class members and functions explicitly marked to be inlined. When the `-no-inline` switch is specified, explicit requests for inlining are ignored.

If the `-auto-inline` switch is specified, the optimizer also inlines some additional functions in cases where the reduction in execution time justifies the increase in code size. [For more information, see “-auto-inline factor” on page 1-22.](#)

When the `-O` switch is specified or implied, the action taken is equivalent to the `-auto-inline` switch.

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.

Interprocedural analysis is enabled by selecting the **Interprocedural analysis** option in the VisualDSP++ **Project Options** dialog box, on the **Compiler** tab, or by specifying the `-ipa` command-line switch.

The `-ipa` switch automatically enables the `-O` switch to turn on optimization. However, all object files supplied in the final link must have been compiled with the `-ipa` switch; otherwise, undefined behavior may result.

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

The `-ipa` optimizations are invoked after the initial link, when the prelinker reinvokes the compiler to perform the new optimizations. Because a file may be recompiled by the prelinker, you cannot use the `-S` option to see the final optimized assembler file when `-ipa` is enabled. Instead, you must 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 the source files to build up usage information about all of the function and data items. The compiler uses the information to make additional optimizations across all of the source files. One of these optimizations removes functions that are never called. This optimization can significantly reduce the overall size of the final executable.

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. Although IPA will generate usage information for potential additional optimizations at the final link stage, neither the usage information nor the module's source file are available when the linker includes a module from a library.

Each library module is compiled to the normal `-O` optimization level, but the prelinker cannot access the previously generated additional usage information for an object in a library. Therefore, IPA cannot exploit the additional information associated with a library module.

If a library module has to make calls to a function in a user module in the program, IPA must be informed that these calls may occur. IPA must be informed of these calls because IPA examines all the visible calls to the function and determines how best to optimize it based on that information received. However, IPA cannot “see” the calls to the function from the library because the library code has no associated usage information to show that it uses the function.

A `pragma, retain_name`, tells IPA there are calls that it cannot see, as shown in the following example.

```
int delete_me(int x) {
    return x-2;
}
#pragma retain_name("keep_me")
int keep_me(int y) {
    return y+2;
}
int main(void) {
    return 0;
}
```

Compiler Command-Line Interface

When this program is compiled and linked with `-ipa`, IPA can see that there are no calls to `delete_me()` in any of the source files (one source file, in this case). Therefore, IPA deletes `delete_me()` as unnecessary. IPA does not delete `keep_me()`, because the `retain_name` pragma informs IPA that there are uses of the function not visible to IPA. No pragma is necessary for `main()`, because IPA knows this is the entry-point to the program.

IPA assumes it can see all calls to a function and uses this knowledge (of the parameters being passed to a function) to effectively tailor the code generated for a function.



If there are calls to a function from an object module in a library, IPA does not have access to the information for that invocation of the function and this may cause it to incorrectly optimize the generated code.

C/C++ Compiler Language Extensions

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

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 brief descriptions of each switch beginning [on page 1-21](#).

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 you are porting a program that uses the extra ADI 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`, you can use `__inline`.

This section provides an overview of the extensions, brief descriptions, and pointers to more information on each extension.

[Table 1-7](#) and [Table 1-8](#) provide descriptions of each extension and direct you to sections that document each extension in more detail.

C/C++ Compiler Language Extensions

Table 1-7. Keyword Extensions

| Keyword Extensions | Description |
|--------------------------------|--|
| <code>inline</code> | Directs the compiler to integrate the function code into the code of its callers. For more information, see “Inline Function Support Keyword (inline)” on page 1-57. |
| <code>asm()</code> | Places Blackfin DSP core assembly language commands directly in your C/C++ program. For more information, see “Inline Assembly Language Support Keyword (asm)” on page 1-58. |
| <code>section("string")</code> | Specifies the section in which an object or function is placed. For more information, see “Placement Support Keyword (section)” on page 1-70. |
| <code>bool, true, false</code> | Specifies a Boolean type. For more information, see “Boolean Type Support Keywords (bool, true, false)” on page 1-71. |
| <code>restrict</code> | Specifies restricted pointer features. For more information, see “Pointer Class Support Keyword (restrict)” on page 1-71. |

Table 1-8. Operational Extensions

| Operational Extensions | Description |
|---------------------------------|--|
| Non-constant initializers | Lets you use non-constants as elements of aggregate initializers for automatic variables. For more information, see “Non-Constant Aggregate Initializer Support” on page 1-72. |
| Indexed initializers | Lets you specify elements of an aggregate initializer in an arbitrary order. For more information, see “Indexed Initializer Support” on page 1-73. |
| Variable length arrays | Lets you create local arrays with a variable size. For more information, see “Variable Length Arrays” on page 1-75. |
| Preprocessor generated warnings | Lets you generate warning messages from the preprocessor. For more information, see “Preprocessor Generated Warnings” on page 1-75. |
| C++ style comments | Allows for “//” C++ style comments in C programs. For more information, see “C++ Style Comments” on page 1-75. |

Inline Function Support Keyword (inline)

The `inline` keyword directs the compiler to integrate the code for the function you declare as `inline` into the code of its callers. `inline` is a standard feature of C++; the `ccblkfn` compiler provides it as a C extension.

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));  
}
```

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

In some cases, the compiler does not output object code for the function; for example, when the address is not needed for an `inline` function which is called from within the defining program. However, recursive calls and functions whose addresses are explicitly referred to by the program are compiled to assembly code.



The `-no-inline` and `-traditional` switches disable function inlining. For more information, see brief descriptions of each switch, beginning [on page 1-21](#).

Inline Assembly Language Support Keyword (asm)

The compiler's `asm()` construct allows you to code Blackfin DSP assembly language instructions within a C/C++ function. Use the `asm()` construct to express assembly language statements that cannot be expressed easily or efficiently with C or C++ constructs.

The `asm()` keyword allows you to code complete assembly language instructions or 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 *does* perform substitutions for operands of the formats `%0` through `%9`. However, it passes *everything else* to the assembler without reading or analyzing it.



The `asm()` constructs, defined before the variable declarations within a function, are flagged as errors because executable statements are not allowed before declarations in the C/C++ code.

A simplified `asm()` construct without operands takes the form of

```
asm("R0=0;");
```

The complete assembly language instruction, enclosed in double quotes, is the argument to `asm()`. Using `asm()` constructs with operands requires some additional syntax. Strictly speaking, the above example needs elaboration to notify the compiler that the register `R0` is overwritten. [For more information, see “Assembly Construct Operand Description” on page 1-62.](#)

The construct syntax is described in:

- [“Assembly Construct Template” on page 1-59](#)
- [“Assembly Construct Operand Description” on page 1-62](#)

- [“Assembly Constructs with Multiple Instructions” on page 1-68](#)
- [“Assembly Construct Reordering and Optimization” on page 1-68](#)
- [“Assembly Constructs with Input and Output Operands” on page 1-69](#)

Assembly Construct Template

Use `asm()` constructs to specify the operands of assembly instruction using C or C++ expressions. You do not need to know which registers or memory locations contain C or C++ variables.

`asm()` Constructs Syntax

Use the following general syntax for your `asm()` constructs.

```
asm(
    template
    [:[constraint(output operand)[,constraint(output operand)...]]
    [:[constraint(input operand)[,constraint(input operand)...]]
    [:clobber]]
);
```

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 1-68](#).

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 1-62](#).

output operand

The output operand is the name of a C or C++ variable that receives output from a corresponding operand in the assembly instruction.

input operand

The input operand is a C/C++ expression that provides an input to a corresponding operand in the assembly instruction.

clobber


The clobber notifies the compiler that a list of registers are 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. See [Table 1-10 on page 1-67](#).

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
- Add a space between adjacent colon field delimiters in order to avoid a clash with the “::” reserved global resolution operator.

 If there are no output operands and there are input operands, you must use two consecutive colons to separate the assembly template from the input operands. These two colons must be separated by a space; otherwise, the two colons will be treated as a C++ namespace identifier.

- 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 DSPs assembly language assignment instruction.

```
{
  int result, x;
  ...
  asm (
```

```
    "%0=%1;" :  
    "=d" (result) :  
    "d" (x)  
    );  
}
```

In the previous example, 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 r 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 different kinds of registers or in memory, the compiler generates code to copy the values into an r register before the asm() construct uses them.

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 1-9 on page 1-66](#) describes the correspondence between constraint letters and register classes.

-  The use of any letter not listed in [Table 1-9](#) 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 =.
- 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 what kind of operand they describe by means of preceding symbols. The possible preceding symbols are: 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 will be loaded with the value of the C/C++ expression before the `asm()` template is executed. Its C/C++ expression will not be modified by the `asm()`, 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 still 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. The operand must appear as part of the second argument to the `asm()` construct. Example: `"#d"`

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.

C/C++ Compiler Language Extensions

For example,

```
asm("%0 += %1 * %2;"
    : "+a0"(sum)      /* output */
    : "H"(x), "H"(y)  /* input */
    );
```

would load `sum` into `A0`, and load `x` and `y` into two `DREG` halves, execute the operation, and then store the new total from `A0` back into `sum`.

[Table 1-9](#) provides the register names. 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 1-10 on page 1-67](#) describes clobber specifiers.



Naming the registers in this way allows the `asm()` to specify several registers that must be related, such as the `DAG` registers for a circular buffer.

Table 1-9. `asm()` Operand Constraints

| Constraint | Register Type | Registers |
|------------|------------------------------|----------------|
| a | General addressing registers | P0 - P5 |
| p | General addressing registers | P0 - P5 |
| i | DSP addressing registers | I0 - I3 |
| b | DSP addressing registers | I0 - I3 |
| d | General data registers | R0 - R7 |
| r | General data registers | R0 - R7 |
| D | General data registers | R0 - R7 |
| A | Accumulator registers | A0, A1 |
| e | Accumulator registers | A0, A1 |
| f | Modifier register | M0 - M3 |
| E | Even general data registers | R0, R2, R4, R6 |

Table 1-9. asm() Operand Constraints (Cont'd)

| Constraint | Register Type | Registers |
|--------------|---|-----------------------|
| 0 | 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 |
| 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 1-10. Clobber String Significance

| Clobber String | Meaning |
|---|-----------------------------|
| "r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7", | General data register |
| "p0", "p1", "p2", "p3", "p4", "p5", "sp", "fp", | General addressing register |
| "i0", "i1", "i2", "i3", | DSP addressing register |
| "m0", "m1", "m2", "m3", | Modifier register |
| "b0", "b1", "b2", "b3", | Base register |
| "l0", "l1", "l2", "l3", | Length register |
| "astat", | ALU status register |
| "seqstat", | Sequencer status register |

Table 1-10. Clobber String Significance (Cont'd)

| Clobber String | Meaning |
|----------------|--------------------------------|
| "rets", | Subroutine address register |
| "cc", | Condition code register |
| "a0", "a1", | Accumulator result register |
| "lc0", "lc1", | Loop counter register |
| "memory" | Unspecified memory location(s) |

Assembly Constructs with Multiple Instructions

There can be many assembly instructions in one template. The input 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.

If the `asm()` string is longer than one line, you may continue it on the next line by placing a backslash (`\`) at the end of the line.

This is an example of multiple instructions in a template:

```
/* (pseudo code) r7 = from; r6 = to; result = from + to; */
asm ("r7=%1; \
r6=%2; \
%0=r6+r7;"
    : "d" (result)           /* output */
    : "d" (from), "d" (to)   /* input */
    : "r7", "r6");          /* clobbers */
```

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 you must be careful to notify the compiler that you are using them by using the clobber specifiers (see [Table 1-10 on page 1-67](#)).

The compiler may eliminate supplied assembly instructions if the output operands are not used, move them out of loops, or replace two with one if they constitute a common subexpression. 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, combined, 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 statement.

Assembly Constructs with Input and Output Operands

The output operands must be write only; the compiler assumes that the values in these operands do not need to be preserved. When the assembler instruction has an operand that is read from and written to, you must logically split its function into two separate operands: one input operand and one write-only output operand. The connection between the two operands is expressed by constraints in the same location when the instruction executes.

When a register's value is to be both an input and an output, and the final value is to be stored to the same location from which the original value was loaded, the register can be marked as an input-output, using the "+" constraint symbol, as described earlier.

C/C++ Compiler Language Extensions

If the final value is to be saved into a different location, then both an input and an output must be specified, and the input must be tied to the output by using its position number as the constraint. 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
```

Assembly Constructs and Flow Control

It is inadvisable to place flow control operations within an `asm()` construct that “leaves” the `asm()` construct, such as calling a procedure or performing a jump, to another piece of code that is not within the `asm()` construct itself. Such operations are invisible to the compiler 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.

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 `section()`’s string literal parameter. If you do not specify a `section()` 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 1-114](#).

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, are explicitly `static`, or are given as external-object definitions. The example below shows the declaration of a static variable that is placed in the section called `bingo`.

```
static section("bingo") int x;
```

Boolean Type Support Keywords (`bool`, `true`, `false`)

The `bool`, `true`, and `false` keywords are extensions that support the C++ boolean type. The `bool` keyword is a unique signed integral type, just as the `wchar_t` is a unique unsigned 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 non-zero 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 automatically converted to `bool` when needed.

These keyword extensions behave as if the declaration that follows had appeared at the beginning of the file, except that assigning a non-zero integer to a `bool` type always causes it to take on the value `true`.

```
typedef enum { false, true } bool;
```

Pointer Class Support Keyword (`restrict`)

The `restrict` keyword is an extension that supports restricted pointer features. The use of `restrict` is limited to the declaration of a pointer and specifies that the pointer provides exclusive initial access to the object to which it points. More simply, `restrict` 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 are not aliases.

C/C++ Compiler Language Extensions

The compiler is free to use the information about restricted pointers and aliasing to better optimize C or C++ code that uses pointers. The `restrict` keyword is most useful when applied to function parameters about which the compiler would otherwise have little information.

```
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.

Non-Constant Aggregate Initializer Support

The compiler includes extended support for aggregate initializers. The compiler does not require the elements of an aggregate initializer for an automatic variable to be constant expressions. The following example shows an initializer with elements that vary at run time.

```
void initializer (float a, float b)
{
    float the_array[2] = { a-b, a+b };
}

void foo (float f, float g)
{
    float beat_freqs[2] = { f-g, f+g };
}
```


All automatic structures can be initialized by arbitrary expressions involving literals, previously declared variables and functions.

Indexed Initializer Support

ANSI/ISO Standard C/C++ requires that the elements of an initializer appear in a fixed order—the same order as the elements in the array or structure being initialized. The `ccblkn` compiler, by comparison, supports labeling elements for array initializers. This feature lets you specify the array or structure elements in any order by specifying the array indices or structure field names to which they apply.

For an array initializer, the syntax `[INDEX]` appearing before an initializer element value specifies the index to be initialized by that value. Subsequent initializer elements are then applied to sequentially following elements of the array, unless another use of the `[INDEX]` syntax appears. The index values must be constant expressions, even if the array being initialized is automatic.

The following example shows equivalent array initializers—the first in standard C and C++ and the next using the extension. Note that the `[index]` precedes the value being assigned to that element.

```
/* Example 1 C Array Initializer */
/* Standard C array initializer */

int a[6] = { 0, 0, 15, 0, 29, 0 };

/* equivalent ccblkn C array initializer */

int a[6] = { [4] 29, [2] 15 };
```

You can combine this technique of naming elements with standard C/C++ initialization of successive elements. The Standard C/C++ and compiler instructions below are equivalent. Note that any unlabeled initial value is assigned to the next consecutive element of the structure or array.

C/C++ Compiler Language Extensions

```
/* Example 2 Standard C & ccblkfn /C++ C Array Initializer */  
/* Standard C array initializer */
```

```
int a[6] = { 0, v1, v2, 0, v4, 0 };
```

```
/* equivalent ccblkfn C array initializer that uses indexed  
elements */
```

```
int a[6] = { [1] v1, v2, [4] v4 };
```

The following example shows how to label the array initializer elements when the indices are characters or an enum type.

```
/* Example 3 C Array Initializer With enum Type Indices */  
/* ccblkfn C array initializer */
```

```
int whitespace[256] =  
{  
[' ' ] 1, ['\t'] 1, ['\v'] 1, ['\f'] 1, ['\n'] 1, ['\r'] 1  
};
```

In a structure initializer, specify the name of the field to initialize with `fieldname:` before the element value. The Standard C/C++ and compiler's C struct initializers in the example below are equivalent.

```
/* Example 4 Standard C & ccblkfn C struct Initializer */  
/* Standard C struct Initializer */
```

```
struct point {int x, y};  
struct point p = {xvalue, yvalue};
```

```
/* Equivalent ccblkfn C struct Initializer With Labeled  
Elements */
```

```
struct point {int x, y};  
struct point p = {y: yvalue, x: xvalue};
```

Preprocessor Generated Warnings

The preprocessor directive `#warning` causes the preprocessor to generate a warning and continue preprocessing. The text on the remainder of the line that follows `#warning` is used as the warning message.

Variable Length Arrays

The `ccblkn` compiler allows variable length arrays to be created on the stack when a function is invoked. Standard C and C++ requires the size of an array to be known at compile time. The following example shows a function that has an array whose size is determined by the value of a parameter passed into the function.

```
void var_array (int nelms, int *ival)
{
    int temp[nelms];
    int i;

    for (i=0;i<nelms; i++)
        temp[i] = ival[i]*2;
}
```

C++ Style Comments

The compiler accepts C++ style comments in C programs, beginning with `//` and ending at the end of the line. This is compatible with standard C, except for:

```
a = b

/* highly unusual */ c
```

which a standard C compiler processes as

```
a = b / c;
```

Built-In Functions

The compiler supports intrinsic functions that enable efficient use of hardware resources. 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 which begin with `__builtin_`. Note that identifiers beginning with double underlines (`__`) 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` option is used.

These functions are specific to individual architectures and this section lists the built-in functions supported at this time on Blackfin DSPs. Various system header files provide you with definitions and access to the intrinsics as described below.

The compiler supports:

- [“Fractional Value Built-Ins” on page 1-76](#)
- [“Viterbi History and Decoding Functions” on page 1-80](#)
- [“Non-Fractional Arithmetic Built-ins” on page 1-82](#)
- [“Circular Buffer Built-In Functions” on page 1-83](#)
- [“System Built-In Functions” on page 1-84](#)

Fractional Value Built-Ins

The built-in functions provide access to the fractional arithmetic and the parallel 16-bit operations supported by the Blackfin DSP instructions. Various C types are defined to represent these classes of data.

Table 1-11. Fractional Value C Types

| C type | Usage |
|----------------------|---------------------------------------|
| <code>fract16</code> | Single 16-bit signed fractional value |
| <code>fract32</code> | Single 32-bit signed fractional value |

Fractional types have a representation similar to integer types, except that the binary point is at the left end, immediately following the sign bit.

Single Fractional Values

The `fract16` type represents a single 16-bit signed fractional value, while 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.

The `fract.h` header file provides access to the definitions for each of the built-in functions that support single fractional values. These functions have names with suffixes `_fr1x16` for single `fract16` and `_fr1x32` for single `fract32`.



The Blackfin DSP compiler does not support 32x32->32 *fractional* multiplication, as opposite of the standard C multiply operation that is 32x32->32 integer. Therefore, `mult_fr1x32` takes two `fract16`s.

The following functions are available:

```
fract16 add_fr1x16(fract16, fract16);
fract16 sub_fr1x16(fract16, fract16);
fract16 mult_fr1x16(fract16, fract16);
fract16 multr_fr1x16(fract16, fract16);
fract32 mult_fr1x32(fract16, fract16);
fract16 abs_fr1x16(fract16);
fract16 min_fr1x16(fract16, fract16);
fract16 max_fr1x16(fract16, fract16);
fract16 negate_fr1x16(fract16);
```

C/C++ Compiler Language Extensions

```
fract16 shl_fr1x16(fract16, int);
fract16 shr_fr1x16(fract16, int);
fract32 add_fr1x32(fract32, fract32);
fract32 sub_fr1x32(fract32, fract32);
fract32 abs_fr1x32(fract32);
fract32 min_fr1x32(fract32, fract32);
fract32 max_fr1x32(fract32, fract32);
fract32 negate_fr1x32(fract32);
fract32 shl_fr1x32(fract32, int);
fract32 shr_fr1x32(fract32, int);
fract16 sat_fr1x32(fract32);
fract16 round_fr1x16(fract32);
int norm_fr1x32(fract32);
int norm_fr1x16(fract16);
```

The fractional arithmetic is saturating. For convenience, if `fract.h` is included with `ETSI_SOURCE` defined, the macros listed below will also be defined, mapping from the European Telecommunications Standards Institute's `fract` functions onto the compiler built-ins.

| | |
|-------------------------|----------------------------|
| <code>add()</code> | <code>L_shr()</code> |
| <code>sub()</code> | <code>L_mult()</code> |
| <code>abs_s()</code> | <code>L_mac()</code> |
| <code>shl()</code> | <code>L_msu()</code> |
| <code>shr()</code> | <code>saturate()</code> |
| <code>mult()</code> | <code>extract_h()</code> |
| <code>mult_r()</code> | <code>extract_l()</code> |
| <code>negate()</code> | <code>L_deposit_l()</code> |
| <code>round()</code> | <code>L_deposit_h()</code> |
| <code>L_add()</code> | <code>div_s()</code> |
| <code>L_sub()</code> | <code>norm_s()</code> |
| <code>L_abs()</code> | <code>norm_l()</code> |
| <code>L_negate()</code> | <code>L_Extract()</code> |
| <code>L_shl()</code> | <code>L_Comp()</code> |
| | <code>Mpy_32()</code> |
| | <code>Mpy_32_16()</code> |

Fractional literals are supported only in C++ mode. However, in C mode, floating-point values may be converted to fractional values, using the built-in functions below. Sometimes, there may be a loss of precision as a result of the conversion to `fract32`, as the float provides only 24 bits of significance.

```
fract16 float_to_fr16(float);
float fr16_to_float(fract16);
fract32 float_to_fr32(float);
float fr32_to_float(fract32);
fract16 fr32_to_fr16(fract32);
fract32 fr16_to_fr32(fract16);
```

When optimizing programs that use `fract16` operations, the compiler automatically detects cases where parallel operations can be performed and issues the appropriate instructions. The `L_mac()` and `L_msu()` functions are implemented as macros that invoke multiplication and addition built-ins; the compiler converts these built-ins to MAC operations when appropriate.

Complex Fractional Built-ins

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;
```

C/C++ Compiler Language Extensions

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 3-25](#).

The compiler also supports the following built-in operations for complex fracts.

- Complex fractional multiply and accumulate and multiply and subtract

```
cmac_fr16();  
cmsu_fr16();
```

- Complex fractional square

```
csqu_fr16();  
csqu_fr32();
```

- Complex fractional distance

```
cdst_fr16();  
cdst_fr32();
```

Viterbi History and Decoding Functions

There are four built-in functions available which provide the selection function of a Viterbi decoder. Specifically, these four functions provide the maximum value selection and history update parts. The functions all make use of 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.)

The four Viterbi functions allow for left or right shifting (setting the least or most significant bit, accordingly), and for 1×16 or 2×16 operands.

The four Viterbi functions are multi-valued; they update some of their parameters inplace, since Viterbi operations return both the selection result and the updated history. The first two Viterbi functions provide left- and right-shifting operations for single 16-bit input operands. The functions are:

```
void lvitmax1x16(int value, int oldhist,
                short selected, int newhist)
void rvitmax1x16(int value, int oldhist,
                short selected, int newhist)
```

The two functions, `lvitmax1x16()` and `rvitmax1x16()`, perform “selection-and-update” operations for two 16-bit operands, which are in the high and low halves of “value”, respectively. The “oldhist” operand contains the history value from the preceding iteration. The “selected” and “newhist” operands are not inputs at all; instead, their expressions must be lvalues (valid on the left-hand side of an assignment), whose values are updated by the operation.

The “selected” is set to contain the largest half of value. The “newhist” is set to contain “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 second two Viterbi functions provide left and right shifting operations for pairs of 16-bit input operands. The functions are:

```
lvitmax2x16(int value_x, int value_y, int oldhist,
            short selected, int newhist)
void rvitmax2x16(int value_x, int value_y, int oldhist,
                short selected, int newhist)
```

The two functions, `lvitmax2x16()` and `rvitmax2x16()`, perform two selection-and-update operations. Each of the `value_x` and `value_y` input expressions contains two 16-bit operands. A selection operation is performed on the two 16-bit operands in `value_x`, and another selection operation is performed on the two 16-bit operands in `value_y`. The `oldhist` 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 `value_x` selection operation. Then, the history value is shifted a second place, and another bit is set to indicate the result for `value_y` selection operation.

The selected value from `value_x` is stored into the low half of `selected`.
The selected value from `value_y` is stored into the high half of `selected`.

Non-Fractional Arithmetic Built-ins

The `ccblkfn` compiler's non-fractional built-in functions provide access to miscellaneous operations supported by the Blackfin DSP instructions. The following functions provide the partial divide primitives `DIVS` and `DIVQ`.

```
int divs(int x, int y);  
int divq(int x, int y);
```

The following functions provide exponent detection on single or parallel values.

```
int expadj1x32(int x, short y);  
int expadj2x16(int x, short y);  
int expadj1x16(short x, short y);
```

The following function returns the number of bits which are set to 1.
Available through `math.h`.

```
int countones(int)
```

Circular Buffer Built-In Functions

The C/C++ compiler provides the following two built-in functions for using the Blackfin DSP's circular buffer mechanisms.

Circular Buffer Increment of an Index

The following operation performs a circular buffer increment of an index.

```
int circindex(int index, int incr, int nitems)
```

The operation is:

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

Circular Buffer Increment of a Pointer

The following operation

```
void *circptr(void *ptr, size_t incr, void *base, size_t buflen)
```

performs a circular buffer increment of a pointer. Both *incr* and *buflen* are specified in bytes, since the operation deals in void pointers. The operation is:

```
ptr += incr;
if (ptr < base) ptr += buflen;
else if (ptr >= (base+buflen)) ptr -= buflen;
```

When the `-circbuf` switch (see [on page 1-24](#)) is specified, the compiler also attempts to generate circular buffer increments for modulus array references such as `array[index % nitems]`. For this to happen, the compiler must be able to determine the starting value for `index` will be within the range `0..(nitems-1)`.

In addition, the `-circbuf` switch tells the compiler to treat array references of the form `"array[i%n]"` as a circular buffer operation on the array.

System Built-In Functions

The following built-in functions allow access to system facilities on the Blackfin DSP. The functions are all defined in `<ccblkfn.h>`.

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

```
int sysreg_read(reg)
int sysreg_write(reg, int val)
```

These functions get (read) or set (write) the value of a system register. In both 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 into the idle mode.

Synchronization

```
void csync(void)
void ssync(void)
```

These two functions provide synchronization. `csync()` is a core-only synchronization—it flushes the pipeline and store buffers. The `ssync()` is a system synchronization, and also waits for an ACK from the system bus.

Pragmas

The Blackfin DSP's C/C++ compiler supports a number of pragmas. Pragmas are implementation-specific directives that modify the compiler's behavior. The C/C++ compiler supports pragmas for:

- Arranging special alignment for data
- Defining functions that act as interrupt or exception handlers
- Giving additional information about loop usage to improve optimization
- Changing the optimization level, midway through a module
- Changing how an externally visible function is linked

The following sections describe the pragmas that support the features listed above.

- [“Data Alignment Pragmas”](#)
- [“Interrupt Handler Pragmas”](#) on page 1-88

- [“Loop Optimization” on page 1-88](#)
- [“General Optimization” on page 1-89](#)
- [“Linking Pragma Behavior” on page 1-90](#)
- [“Pragmas to Assist in Vectorizing Loops” on page 1-90](#)

Data Alignment Pragmas

The data alignment pragmas are used to modify how the compiler arranges data within memory. Since the Blackfin DSP architecture requires memory accesses to be naturally aligned, each data item is normally aligned at least as strongly as itself—two-byte shorts have alignment of 2, and four-byte longs have alignment of 4.

When structs are 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 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 reduced, so that a large array occupies less space. A struct may have its alignment increased to improve the compiler’s opportunities in vectorizing access to the data.

The data alignment pragmas include `ALIGN NUM`, `PACK (ALIGNOPT)`, and `PAD (ALIGNOPT)` pragmas. Alignments specified using these pragmas must be a power of 2.

ALIGN NUM Pragma

The `align num` 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 should be forced to be aligned on a boundary specified by `num`.

- If `num` is greater than the alignment normally required by the following variable or field declaration, then the variable or field declaration's alignment is changed to be `num`.
- If `num` is less than alignment normally required, then the variable or field declaration's alignment is changed to be `num`, and a warning is given that the alignment has been reduced.

If the `pack` or `pad` pragmas (see below) are currently active then `align` will override for the immediately following field declaration.

PACK (ALIGNOPT) Pragma

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`.

The pragma is used to reduce the default alignment of the struct to be aligned. If there are fields within the struct that have a default alignment greater than `align`, their alignment is reduced to be `alignopt`. If there are fields within the struct that have alignment less than `align`, their alignment is unchanged.

If `alignopt` is specified, it is illegal to invoke `#pragma pad`, until default alignment is restored.

PAD (ALIGNOPT) Pragma

The `pad()` pragma may be applied to struct definitions. It applies to struct definitions that follow, until the default alignment is restored by omitting `alignopt`.

This pragma is effectively shorthand for placing `#pragma align` before every field within the struct definition. Like `pragma pack`, it reduces the alignment of fields which default to an alignment greater than `alignopt`. However, unlike `pragma pack`, it also increases the alignment of fields which default to an alignment less than `alignopt`.



While `pragma alignopt` generates a warning if a field alignment is reduced, `pragma pad alignopt` does not.

If `alignopt` is specified, it is illegal to invoke `#pragma pack`, until default alignment is restored.

Interrupt Handler Pragas

The `interrupt`, `nmi`, and `exception` pragmas all declare that the following function declaration or definition is to be used as an entry in the Event Vector Table. The compiler arranges for the function to save its context above and beyond the usual caller-preserved set of registers, and to restore the context upon exit. The function will return using an instruction appropriate to the type of event specified by the pragma.

These pragmas are not normally used directly; these are macros provided by the `<sys/exception.h>` file. See [“Interrupt Handler Support” on page 1-99](#) for more information.

The pragmas may be specified on either the 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 in conjunction 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.

Loop Optimization

Loop pragmas give the compiler additional loop usage information, which allows the compiler to perform more aggressive optimization. The pragmas are placed before the loop statement, and apply to the statement that immediately follows. In general, it is most effective to apply loop pragmas to inner-most loops, since the compiler can achieve the most savings there.

Loop pragmas are:

- **VECTOR_FOR Pragma**

Use this pragma to tell the compiler that each iteration of the loop does not depend on values computed during the previous iteration. The pragma helps the compiler to execute two iterations of the loop in parallel.

- **NO_ALIAS Pragma**

Use this pragma to tell the compiler 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”, then 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.

With this pragma, you can tell the compiler that the accesses will never conflict, therefore, helping the compiler to perform two iterations in parallel.

General Optimization

The `optimize_off`, `optimize_for_space`, and `optimize_for_speed` pragmas change the optimization level while a given module is being compiled. The `optimize_off` pragma turns off the optimizer, if it was enabled.

The `optimize_for_space` and `optimize_for_speed` pragmas turn the optimizer back on, if it was disabled, or switch focus between reducing code size and increasing performance, if it was already enabled.

Linking Pragma Behavior

Linking pragmas change how a given global function or variable is viewed during the linking stage. These pragmas apply to the following declarations.

- **LINKAGE_NAME *Identifier***

The `linkage_name` pragma associates the *identifier* with the next external function declaration. It ensures that identifier is used as the external reference, instead of following the compiler's usual conventions.

- **RETAIN_NAME Pragma**

The `retain_name` pragma indicates that the external function or variable declaration that follows the pragma is not removed even though Interprocedural Analysis (IPA) sees that it is not used. Use this pragma for C functions that are only called from assembler routines, such as the startup code sequence invoked before `main()`.

Pragmas to Assist in Vectorizing Loops

The optimizer always attempts to vectorize loops when it is safe to do so. The optimizer exploits the information generated by Interprocedural Analysis ([“Interprocedural Analysis” on page 1-52](#)) to increase the cases where it knows it is safe to do so. 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, say `copy(a,b)` and, later, `copy(b,c)`, the Interprocedural Analysis will not be able 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 `vector_for` pragma can be used to explicitly notify the compiler that this is the case. The following is an example of how the pragma should be used.

```
void copy(short *a, short *b) {
    int i;
    #pragma vector_for
    for (i=0; i<100; i++)
        a[i] = b[i];
}
```


The 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 it is 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.

In cases where vectorization is impossible (for example, if array `a` were aligned on a word boundary, but array `b` was not), the information given in the assertion made by `vector_for` may still be put to good use in aiding other optimizations.

Another useful loop pragma is `no_alias`. It asserts that in the next loop, no `load` or `store` 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 any other, iteration of the loop. In the example

above, if the pointers `a` and `b` point to two memory areas that do not overlap, then no load from `b` will be using the same address as any store to `a`. Therefore, `a` is never an alias for `b`.

Using the `no_alias` pragma can lead to better code because it allows any number of iterations to be performed concurrently (rather than just two at a time), thus providing better software pipelining by the optimizer.

 Loop pragmas should only be used on single-block inner-most loops which use the `for` loop construct.

Blackfin DSP Specific Functionality

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

Default Startup Code

The Blackfin C compiler comes with default startup code, which is invoked when the processor starts running. The code initializes a default environment before calling `main()`. The source for this startup code is in the file `VisualDSP\Blackfin\lib\src\libc\basicrt.s`. You may want to modify this code to suit your specific target environment, or perhaps even replace it completely. The `basicrt.s` file contains a number of configuration options that are used to produce the various `crt*.doj` files in the `VisualDSP\Blackfin\lib` directory. The default `.LDF` files link in one of these `crt*.doj` files according to the options specified at link time. See the `basicrt.s` file itself for more details.

Support for argv/argc

You can specify arguments that get passed to your `main()` function when running your program on the simulator. The simulator passes arguments to the program by copying the argument strings into a memory section reserved for this purpose in the Blackfin DSP memory. The memory section's name is `MEM_ARGV`, and its size and location are defined by the `.LDF` file.

Once the simulator has placed the argument strings into the Blackfin DSP memory, the startup code used by the compiler can retrieve the strings and make them available to `main()`.

Use the following steps when passing arguments to your `main()` function using the VisualDSP++ simulator.

Blackfin DSP Specific Functionality

- Enter the arguments to be passed into the **Settings->Simulator->CommandLine Arguments->Command Line Arguments** field. Separate each argument by a comma.

`arg1,arg2,arg3`

- Specify the starting location of the `MEM_ARGV` section in the **Settings->Simulator->Command Line Arguments->Command Line Arguments Base Address** field.

The VisualDSP++ simulator knows the default addresses for `MEM_ARGV` for the standard `.LDF` files. Therefore, under normal circumstances, the default value for this field is acceptable. However, if you modify your LDF, you must ensure that this field holds the correct starting address for `MEM_ARGV`. If this field does not hold the correct address, you may find that your program or data is corrupted when the simulator copies argument strings into Blackfin memory space.

File I/O Support

The VisualDSP++ environment provides access to files on a host system, using `stdio` functions. Because of the hosted nature of the I/O system, there are some limitations on the available functionality:

- Seeking is not supported, so `fseek()` and `rewind()` will fail.
- Input from `stdin` is not supported.

File I/O support is provided through a set of low-level primitives that implement the open, close, read and write operations required. The `stdio` library makes use of these primitives to provide buffered, formatted I/O. If the processor is operating in supervisor mode, the `stdio` library calls these primitives directly. If the processor is operating in user mode, the `stdio` library invokes the primitives through exception 5:

Table 1-12. File I/O Access Steps

| Supervisor mode | User mode |
|---|---|
| Application calls <code>printf()</code> | Application calls <code>printf()</code> |
| <code>printf()</code> formats data into a buffer | <code>printf()</code> formats data into a buffer |
| <code>stdio</code> writes the buffer | <code>stdio</code> raises exception 5 |
| | Exception handler selects the I/O system |
| The I/O system selects the host system | The I/O system selects the host system |
| The output primitive passes the buffer to the host system | The output primitive passes the buffer to the host system |
| The host system displays the string | The host system displays the string |

The source files for the startup code, exception handler and I/O primitives are all available under `...\VisualDSP\Blackfin\lib\src\libc`.

Refer to “[stdio.h](#)” on page 2-11 for more information.

Profiling with Instrumented Code

The profiling facilities allow you to 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. 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 should not be confused with similar functionality in the simulator, which works on a per-instruction basis, rather than a per-function basis.

Generating Instrumented Code

The `-p[1|2]` switch (see [on page 1-35](#)) 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 will write accumulated profile data to the file `"mon.out"` in the current directory.
- The `-p2` option will write accumulated profile data to standard output.
- The `-p` option will write accumulated profile data to both standard output and the `mon.out` file in the current directory.

Linking the Executable

There are two ways to link the executable for profiling: by using `ccblkfn` or by invoking the linker explicitly.

Linking with `ccblkfn` Compiler

When using `ccblkfn`, the `-p[1|2]` switch options also apply to the linking stage. In the simplest case, the source files can be compiled and linked in a single stage.

```
ccblkfn -p -o prog.dxe file1.c file2.c file3.c
```

The files can also be compiled separately, and then linked as

```
ccblkfn -p -c file3.c  
ccblkfn -p -o prog.dxe file1.dox file2.dox file3.dox
```

Linking Explicitly with the Linker

If you invoke the linker explicitly, you need to tell the linker that it is linking a program with profiling enabled. It is not simply a matter of including the profiling library, because a different startup file needs to be

included. The `-MDUSE_PROFILER`, `-MDUSE_PROFILER1` or `-MDUSE_PROFILER2` directives are used to tell the linker that profiling is required (they correspond to `-p`, `-p1` or `-p2`, respectively). For example,

```
linker -o prog.dxe -MDUSE_PROFILER -DADSP-21535 -jcs21
file1.doj file2.doj file3.doj
```

Running the Executable

The executable may produce two forms of output. The first is a dump of data to standard output once the program completes (generated by `-p` and `-p2`). 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 include the sum total of cycle counts in any function invoked from this specified function.

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.

Post-Processing mon.out File

The `profblkfn.exe` program is a Windows program that processes the contents of the `mon.out` file. It reads both the `mon.out` file and the `.DXE` file that produced it. It displays the cycle counts along with the names of the functions recorded in the `mon.out` file associated with the counts. 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 have been produced by the named `.DXE`.

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, 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.

Controlling Available Memory Size

The heap size is specified in the .LDF file in the `VisualDSP\Blackfin\LDF` directory. The compiler uses the `adsp-21532.ldf` file by default. The entry controlling the heap has a similar format to

```
MEM_HEAP { TYPE(RAM) START(0xFF804000) END(0xFF807DFF) WIDTH(8) }
```

The actual values specified in the .LDF file should reflect the memory map available on the actual system.

Internally, `malloc()` uses the `_Sbrk()` library function to obtain additional space from the `HEAP` system. The start and end addresses of the `HEAP` segment can be changed to give a larger or smaller heap, and the library will adjust accordingly. If the segment size is increased, the surrounding segments must be decreased accordingly; otherwise, memory corruption may occur.

Interrupt Handler Support

The Blackfin C/C++ compiler provides support for interrupts and other events used by the Blackfin DSP architecture.

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 1-13. Blackfin System Events

| Event | Priority | Supported |
|------------|----------|-----------|
| Emulation | Highest | No |
| Reset | | Yes |
| NMI | | Yes |
| Exception | | Yes |
| Interrupts | Lowest | Yes |

Resets are supported by treating a reset like a general-purpose interrupt for code generation purposes. This means that the C/C++ compiler supports interrupt, exception and NMI events.



ISRs generated by the C compiler are not currently allowed in the VDK.

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 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, the 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 which are re-entrant, and which re-enable the interrupt system soon after entering the ISR. There is a different macro for specifying a re-entrant interrupt handler:

```
EX_REENTRANT_HANDLER
```

For example, the following code

```
#include <sys/exception.h>  
static int number_of_interrupts;  
  
EX_INTERRUPT_HANDLER(my_isr)  
{  
    number_of_interrupts++;  
}
```

declares and defines `my_isr()` to be a handler for interrupt-type events (for example, the routine returns using an `RETI` instruction). The macro used for defining the ISR is also suitable for declaring it, as a prototype:

```
EX_INTERRUPT_HANDLER(my_isr);
```

Registering an ISR

ISRs, once defined, can be registered in the Event Vector Table (EVT) using the `register_handler()` function. This function operates in a similar manner to the UNIX `signal()` function.

It takes two parameters, defining the event and the ISR, and returns the previously registered ISR, if any. The event is specified using the `interrupt_kind` enumeration from `exception.h`.

For example,

```
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(interrupt_kind, ex_handler_fn);
```

There are two special values that can be passed to `register_handler()` in place of real ISRs:

- `EX_INT_IGNORE` installs a handler that “ignores” the event and immediately returns from the event.
- `EX_INT_DEFAULT` installs the default handler. Currently, the default handler assumes that any event is unexpected and indicative of an error. It prints the system state and aborts the processor with the default handler invoking the current behaviour for ANSI C signals, as described below.

ISRs and ANSI C Signals

ISRs provide similar functionality to ANSI C signal handlers, and their behaviour is related.

An ISR is a function that can be registered directly in the processor's Event Vectot Table (EVT). It 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 will not save or restore the context, nor will it terminate with the correct return instruction. An ISR cannot act as a signal handler, because it will terminate 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 will invoke the signal handler when the event occurs. When the `raise()` function is used to invoke a signal handler explicitly, `raise()` will actually generate the corresponding event, if possible. This will cause the ISR to invoke the signal handler.

You may choose to install normal C functions as signal handlers, or to register ISRs directly, but you should not do both for a given event.

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 and for the values of all the registers that were saved during the prologue. For a list of saved registers, see [“Fetching Saved Registers” on page 1-104](#).

There are two facilities for gaining access to the event context:

- `get_interrupt_info()` function
- `SAVE_REGS()` macro (see [“Fetching Saved Registers” on page 1-104](#))

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 `EXCAUSE` and addresses that caused faults for exceptions. Note that at present, the function needs to be told which kind of event it is investigating.

The structure contains:

```
interrupt_kind kind;
int value;
void *pc;
void *addr;
unsigned status;
interrupt_regs regs;
interrupt_regs *regsaddr;
```

These fields are set as:

- **Exceptions**
The `pc` is set to the value of `RETX`, and `value` is set to the value of `SEQSTAT`.

For exceptions that involve address faults, address and status are set

Blackfin DSP Specific Functionality

to the values of the Memory Mapping Registers (MMRs) for `DATA_FAULT_ADDR` and `DATA_FAULT_STATUS` or `CODE_FAULT_ADDR` and `CODE_FAULT_STATUS`, as appropriate.

- **Hardware Errors**

The `pc` is set to the value of `RETI`, and `value` is set to the value of `SEQSTAT`.

- **NMI Events**

The `pc` is set to the value of `RETN`.

- **All Other Events**

The `pc` is set to the value of `RETI`.

Fetching Saved Registers

The following macro obtains a copy of the registers saved during the ISR prologue:

```
SAVE_REGS(interrupt_info *)
```

It also sets `regsaddr` in the interrupt to point to the start of the saved registers on the supervisor stack. Therefore, any changes made using `regsaddr` within the ISR will be reflected in the processor state when it is restored by the ISR epilogue.

The following registers are always saved during the ISR prologue. They are accessible through the saved context.

- All DREGS (R0, R1, R2, R3, R4, R5, R6, R7)
- All PREGS (P0, P1, P2, P3, P4, P5)
- Frame pointer (FP)
- Arithmetic status (ASTAT)

Additional registers are saved as required, depending on the resources used by the ISR. These registers are not accessible through the saved context. The registers that are optionally saved include:

- Hardware loop registers (LB0, LB1, LT0, LT1, LC0, LC1)
- Accumulators (A0w, A1w, A0x, A1x)
- Condition code register (CC)
- Circular buffer registers (I0-3, L0-3, B0-3, M0-3).

User-Mode Configuration

The default startup code invokes `main()` in supervisor mode, allowing full access to system resources. There is also an alternative configuration which places the Blackfin DSP into the user mode before invoking `main()`. This mode can be invoked using the linker flag `-MDUSERMODE`.

The user-mode configuration installs a default event handler to support operations such as terminating, File I/O, and registering other event handlers.

Allocated Events in User-Mode Configuration

The Blackfin DSP architecture defines sixteen values of `EXCAUSE` for user-level events. Several values are already allocated for various purposes, as described in [Table 1-14](#). The values are defined in the header file `<sys/excause.h>`.

Table 1-14. Allocated Events

| Value | Mnemonic | Description |
|-------|---------------|---|
| 0x0 | EX_EXIT_PROG | Halts the processor. R0 contains exit value. |
| 0x1 | EX_ABORT_PROG | Aborts the processor. R0 contains exit value. |

Blackfin DSP Specific Functionality

Table 1-14. Allocated Events (Cont'd)

| Value | Mnemonic | Description |
|-------|------------|---|
| 0x2 | EX_SYS_REQ | Requests system service. R0 contains a command, R1 and R2 are arguments. R0 contains result, on exit. |
| 0x5 | EX_FILE_IO | Requests File I/O. R4 contains command. R0 to R2 contain arguments. R0 contains result, on exit. |

Table 1-15 lists values for system requests.

Table 1-15. System Requests

| Mnemonic | Description |
|---------------------|---|
| EX_SYSREQ_NONE | Does nothing. |
| EX_SYSREQ_ISR | Registers an ISR in the EVT. R0 is EVT entry (0 to 15). R1 is address of ISR. Returns previous entry in R0. |
| EX_SYSREQ_RAISE_INT | Causes an interrupt. R0 = interrupt number (0-15). |

Table 1-16 lists the values for File I/O.

Table 1-16. File I/O Values

| Mnemonic | Description |
|-----------------|---|
| EX_FILEIO_OPEN | R0 = device, R1 = path, R2 = mode. fid => R0. |
| EX_FILEIO_CLOSE | R0 = fid. |
| EX_FILEIO_WRITE | R0 = fid, R1 = data, R2 = length. Amount => R0. |
| EX_FILEIO_READ | R0 = fid, R1 = data, R2 = length. Amount => R0. |
| EX_FILEIO_SEEK | R0 = fid, R1 = offset, R2 = mode. |
| EX_FILEIO_DUP | R0 = fid. R0 => new fid. |

C/C++ Preprocessor Features

Several features of the C/C++ preprocessor are used by VisualDSP++ to control the programming environment. They are:

- [“Predefined Macros”](#)
- [“Preprocessing of .IDL Files” on page 1-109](#)
- [“Header Files” on page 1-110](#)

The `ccblkn` 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 1-75](#) and [“C++ Style Comments” on page 1-75](#). For ways to write macros, refer to [“Writing Preprocessor Macros” on page 1-110](#).

Predefined Macros

The `ccblkn` compiler defines a number of macros to provide information about the compiler, source file, and options specified. These macros can be tested, using the `#ifdef` and related directives, to support your program’s needs. Similar tailoring is done in the system header files.

Macros such as `__DATE__` can be useful to incorporate in text strings. The `#` operator with a macro body is useful in converting such symbols into text constructs.

[Table 1-17](#) describes the predefined macros.

Table 1-17. Predefined Macro Listing

| Macro | Function |
|-----------------------|--|
| __ADSPBLACKFIN__ | Always defines __ADSPBLACKFIN__ as 1. |
| ADSPBLACKFIN | Defines ADSPBLACKFIN as 1, unless you compile with <code>-pedantic</code> , or <code>-pedantic-errors</code> . |
| __ANALOG_EXTENSIONS__ | Defines __ANALOG_EXTENSIONS__ as 1, unless you compile with <code>-pedantic</code> , <code>-pedantic-errors</code> , or <code>_ansi</code> . |
| __cplusplus | Defines __cplusplus to be 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 <code>mm dd yyyy</code> (ANSI standard). |
| __ECC__ | Always defines __ECC__ as 1. |
| __EDG__ | Always defines __EDG__ as 1. This definition signifies that an Edison Design Group front end is being used. |
| __EDG_VERSION__ | Always defines __EDG_VERSION__ as an integral value representing the version of the compiler's front end. |
| __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 <code>#include</code> command (ANSI standard). |
| __LINE__ | The preprocessor expands this macro into the current input line number as a decimal integer constant (ANSI standard). |
| __NO_BUILTIN | Defines __NO_BUILTIN as 1 when you compile with the <code>-no-builtin</code> command-line switch. |
| _NO_LONGLONG | Always defined _NO_LONGLONG as 1. This definition signifies no support is present for the long long int type. |
| __STDC__ | Always defines __STDC__ as 1, unless you compile with <code>-traditional</code> (ANSI standard). |

Table 1-17. Predefined Macro Listing (Cont'd)

| Macro | Function |
|-------------------------------|--|
| <code>__STDC_VERSION__</code> | Always defines <code>__STDC_VERSION__</code> as 199409L. Undefines this macro if you compile with <code>-traditional</code> (ANSI standard). |
| <code>__TIME__</code> | The preprocessor expands this macro into the preprocessing time as a string constant. The date string constant takes the form <code>hh:mm:ss</code> (ANSI standard). |

Preprocessing of .IDL Files

Every VisualDSP++ Interface Definition Language (VIDL) specification is analyzed by the C++ language preprocessor prior to syntax analysis.

The `#include` directive is used to control the inclusion of additional VIDL source text from a secondary input file that is named in the directive. Two available forms of `#include` are shown in [Figure 1-2](#).

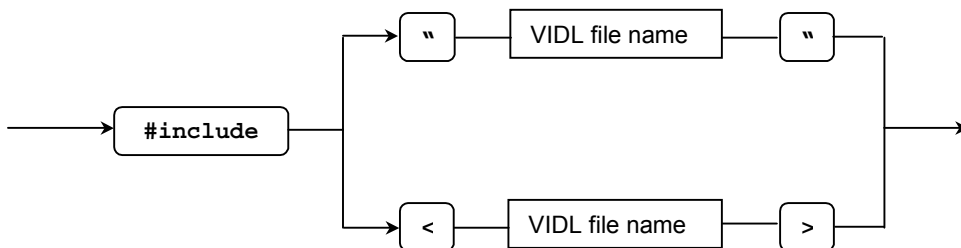


Figure 1-2. #INCLUDE Syntax Diagram

The file identified by the file name is located by searching a list of directories. When the name is delimited by quote characters, the search begins in the directory containing the primary input file, then proceeds with the list of directories specified by the `-I` command-line switch. When the name is delimited by angle-bracket characters, the search proceeds directly with

the directories specified by `-I`. If the file is not located within any directory on the search list, the search may be continued in one or more platform dependent system directories.

Header Files

A header file contains C or C++ declarations and macro definitions. Use the `#include` preprocessor directive to access header files in your program. Header file names have an `.h` or no extension. There are two main categories of header files:

- System header files declare the interfaces to the parts of the operating system. Include these header files in your program for the definitions and declarations you need to access system calls and libraries. Use angle brackets to indicate a system header file. For example, `#include <file>`.
- User header files contain declarations for interfaces between the source files of your program. Use double quotes to indicate a user header file. For example, `#include "file"`.

Writing Preprocessor Macros

A macro is a name of a block of text that the preprocessor substitutes. 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 Statements as Macros

When writing macros, define a macro that expands into a compound statement. You can define such a macro to invoke it the same way you would call a function, making your source code easier to read and maintain.

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 into a compound statement to expanding into a single statement. With the macro expansion into a compound statement, you must sometimes omit the semicolon after the macro call in order to have a valid 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.

With the `do {...} while (0)` construct, you can pretend that the macro is a function and put the semicolon after it.

C/C++ Preprocessor Features

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); /* semicolon from SKIP_SPACES (...); */
else ...
```

Without the `do {...} while (0)` construct, the expansion would be:

```
if (*p != 0)
{
    ...
}
; /* semicolon from SKIP_SPACES (...); */
else
```

For more information on macros, see the *VisualDSP++ 3.0 Assembler and Preprocessor Manual for Blackfin DSPs*.

C/C++ Run-Time Model and Environment

This section provides a full description of the Blackfin DSP run-time model and run-time environment. The run-time model, which applies to compiler-generated code, includes descriptions of the 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 DSPs. Assembly routines linked to C/C++ routines must follow these conventions.



ADI recommends that assembly programmers maintain stack conventions.

[Figure 1-3](#) provides an overview of the run-time environment issues that you must consider as you write assembly routines that link with C/C++ routines. The run-time environment issues include the following items.

- Register usage conventions
 - [“Dedicated Registers” on page 1-116](#)
 - [“Call Preserved Registers” on page 1-116](#)
 - [“Scratch Registers” on page 1-117](#)
 - [“Stack Registers” on page 1-117](#)
- Memory usage conventions
 - [“Using Memory Sections” on page 1-114](#)
 - [“Using Data Storage Formats” on page 1-124](#)
- Program control conventions
 - [“Managing the Stack” on page 1-117](#)
 - [“Transferring Function Arguments and Return Value” on page 1-121](#)

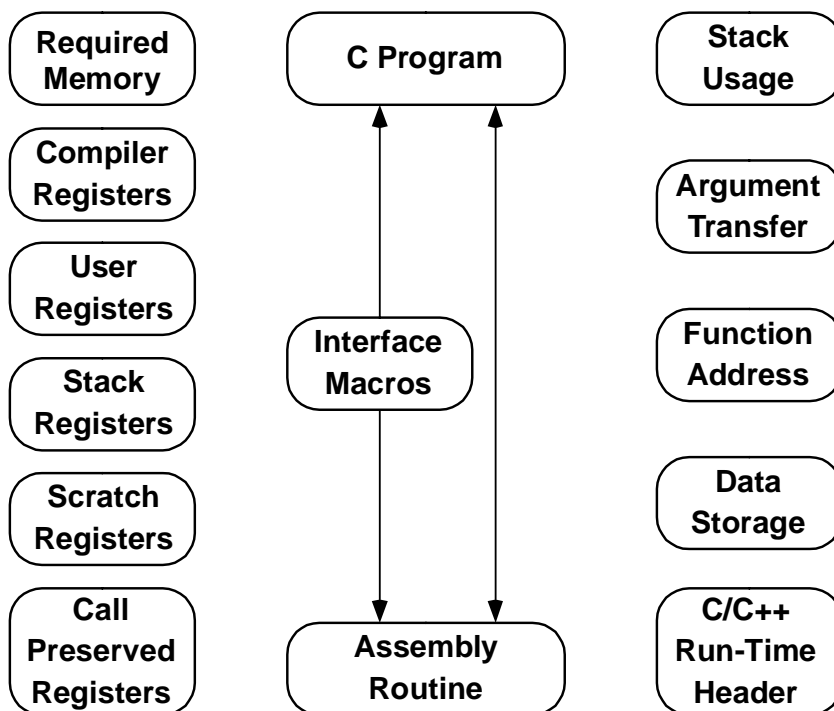


Figure 1-3. Assembly Language Interfacing Overview

Using Memory Sections

The C/C++ run-time environment requires that a specific set of memory section names are used for placing 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 the LDF syntax and other information on the linker, see the *VisualDSP++ 3.0 Linker and Utilities Manual for Blackfin DSPs*.

Code Storage—The code section, `program`, is where the compiler puts all the program instructions that it generates when compiling the program.

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`”.

Run-Time Stack—The run-time stack sections, `stack` and `sysstack`, are where the compiler puts the run-time stack in memory. The processor starts off in supervisor mode, which uses `sysstack` for its run-time stack. You may choose to link your application so the processor is switched to user mode during startup (see [“Default Startup Code” on page 1-93](#)). When in user mode, the run-time stack is in the `stack` section. When linking, use your `.LDF` file to map this section. Because the run-time environment cannot function without this section, you must define it.

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 1-117](#) 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 five functions:

```
malloc()
calloc()
realloc()
free()
```

These functions allocate from the `heap` section of memory by default. The `.LDF` file must define `ldf_heap_space`, `ldf_heap_end` and `ldf_heap_length` to allow the `_Sbrk()` function to determine the location of the heap. Default values for these are defined in the default `.LDF` file.

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. Dedicated registers are:

SP (P6) — FP (P7)
L0 — L3

SP (P6) and FP (P7) are the Stack Pointer and the Frame Pointer Registers, respectively. The compiler requires that both registers are 4-byte aligned and pointing to valid areas within the stack section.

L0 — L3 define the lengths of the DAG's circular buffers. When running compiled code, they must be set to zero, since the compiler uses the DAG registers in non-circular buffer mode.

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 do not need to be saved and restored. Note that the contents of these registers are not preserved across function calls. Scratch registers are:

| | |
|-----------|---|
| 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. |
| LC0 — LC1 | |
| CC | |
| A0 — A1 | |
| I0 — I3 | |
| B0 — B3 | |
| M0 — M3 | |

Stack Registers

The C/C++ run-time environment reserves a set of registers for controlling the run-time stack. These registers may be modified for stack management, but must be saved and restored. Stack registers are:

| | |
|---------|-----------------|
| SP (P6) | — Stack pointer |
| FP (P7) | — 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 1-4 shows an example section of a run-time stack. In the figure, 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 the sequence of steps:

- **Linking Stack Frames** — The return address and the caller's FP are saved on the stack frame, FP is set pointing to the beginning of the frame stack, space for local variables and compiler temporaries is allocated on the frame stack, and the stack pointer is set pointing to the top of the stack frame.
- **Register Saving** — Any registers that the function needs to pre-serve are saved on the stack frame, and the stack pointer is set pointing to the top of the stack frame.

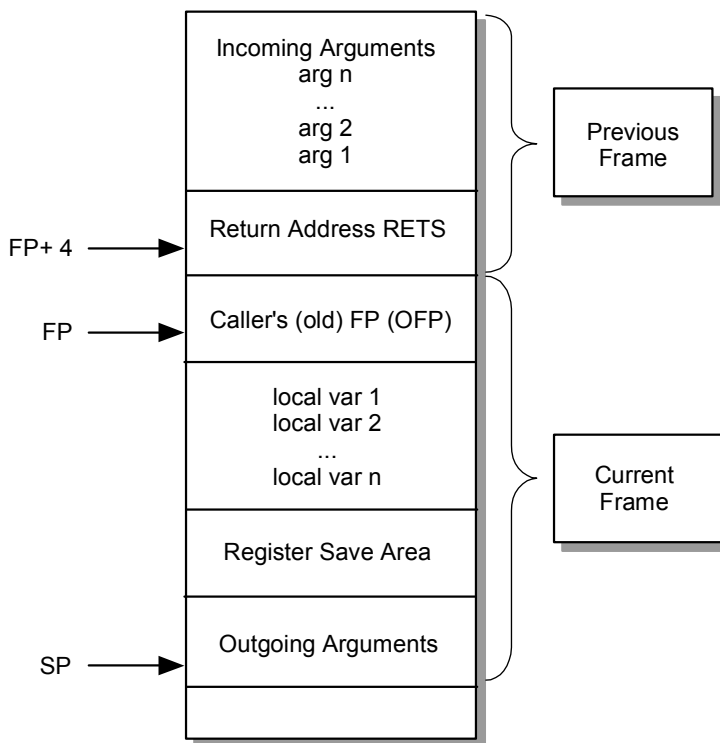


Figure 1-4. Example Run-Time Stack

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 the stack pointer 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 frame pointer; RETS is restored from the stack frame to the return address; and the stack pointer is set pointing to the top of the caller's frame stack.

C/C++ Run-Time Model and Environment

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. You must always allocate at least twelve 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.



The section [“Transferring Function Arguments and Return Value”](#) provides additional detail 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.

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 that have less than twelve bytes of argument data, or that have fewer than three arguments.

Return Values

If a function returns a short or a char, the “callee” is responsible for effecting any sign or zero extension that is needed.

For functions returning aggregate values occupying less than or equal to 32 bits, the result is returned in R0. For aggregate values occupying greater than 32 bits, and less 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.

The callee must copy the return value into the object at the address in P0. [Table 1-18](#) provides examples of passed parameters.

Table 1-18. Examples of Parameter Passing

| Function Prototype | Parameters Passed as | Return Location |
|--|---------------------------------|-----------------|
| <code>int test(int a, int b, int c)</code> | a in R0, b in R1, c in R2 | in R0 |

Table 1-18. Examples of Parameter Passing (Cont'd)

| | | |
|---|---|------------------------------|
| <code>char test(int a, char b, char c)</code> | <code>a in R0, b in R1, c in R2</code> | <code>in R0</code> |
| <code>int test(int a)</code> | <code>a in R0</code> | <code>in R0</code> |
| <code>int test(char a, char b, char c, char d, char e)</code> | <code>a in R0, b in R1, c in R2, d in [FP+20], e in [FP+24]</code> | <code>in R0</code> |
| <code>int test(struct *a, int b, int c)</code> | <code>a (addr) in R0, b in R1, c in R2</code> | <code>in R0</code> |
| <code>struct s2a { char ta; char ub; int vc;} int test(struct s2a x, int b, int c)</code> | <code>x.ta and x.ub in R0, x.vc in R1, b in R2, c in [FP+20]</code> | <code>in R0</code> |
| <code>struct foo *test(int a, int b, int c)</code> | <code>a in R0, b in R1, c in R2</code> | <code>(address) in R0</code> |
| <code>void qsort(void *base, int nel, int width, int (*compare)(const void *, const void *))</code> | <code>base(addr) in R0, nel in R1, width in R2, compare(addr) in [FP+20]</code> | |

Table 1-18. Examples of Parameter Passing (Cont'd)

| | | |
|--|--|---|
| <pre>struct s2 { char t; char u; int v; } struct s2 test(int a, int b, int c)</pre> | <pre>a in R0, b in R1, c in R2</pre> | <pre>in R0 (s.t and s.u) and in R1 (s.v)</pre> |
| <pre>struct s3 { char t; char u; int v; int w; } struct s3 test(int a, int b, int c)</pre> | <pre>a in R0, b in R1, c in R2</pre> | <pre>in *P0 (based on value of P0 at the call, not necessarily at the return)</pre> |

Using Data Storage Formats

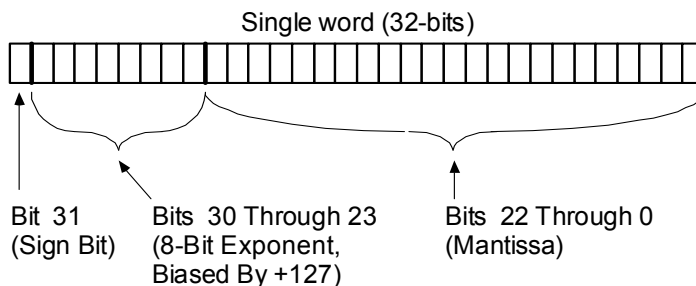
The C/C++ run-time environment uses the data formats that appear in the [Table 1-19](#) and [Figure 1-5](#).

Table 1-19. Data Storage Formats and Data Type Sizes

| Applied Type | Number Representation |
|--------------------|---------------------------|
| char | 8-bit two's complement |
| unsigned char | 8-bit unsigned magnitude |
| short int | 16-bit two's complement |
| unsigned short int | 16-bit unsigned magnitude |
| int | 32-bit two's complement |
| unsigned int | 32-bit unsigned magnitude |
| long int | 32-bit two's complement |
| unsigned long int | 32-bit unsigned magnitude |

Table 1-19. Data Storage Formats and Data Type Sizes (Cont'd)

| Applied Type | Number Representation |
|--------------|------------------------------|
| float | 32-bit IEEE single-precision |
| double | 32-bit IEEE single-precision |



The single word (32-Bit) data storage format equates to:

$$-1^{\text{Sign}} \times 1.\text{Mantissa} \times 2^{(\text{Exponent} - 127)}$$

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 |

Figure 1-5. Data Storage Format for Float and Double Types

Basic Startup Code Sequence

The basic startup code is an assembly language procedure that initializes the processor and sets up processor features to support the C run-time environment. The source code for the default run-time header is in the `basiccrt.s` file.

This run-time initialization code performs these operations:

1. Resets registers.
2. Initializes Event Vector Table.
For supervisor mode, the startup code installs a vector for IVG15, so that the processor can be switched to lowest supervisor priority.

For user mode, the startup code installs an exception handler for File I/O and similar requests. All other entries are cleared.
3. Sets up a stack pointer, including user-mode stack pointer if needed, and enables the cycle counter.
4. Initializes File I/O support.
5. Enables interrupts. For supervisor mode, only IVG15 is enabled; for user mode, all interrupts are enabled.
6. Switches processor mode from the Reset priority to IVG15 (lowest supervisor mode priority) or to user mode.
7. Initializes profiling, if necessary.
8. Initializes C/C++ library internal `mutexs` if multi-threaded support is enabled.
9. Initializes any global C++ objects and records a destruction call for cleanup at program exit.
10. Initializes `argc` and the `argv` array.

11. Calls `main()` to start the actual program.
12. Calls `exit()`.
13. The `atexit` functions flush any accumulated profiling data, and close down File I/O.

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 1-113](#). At the end of this reference, a series of examples demonstrate how to mix C/C++ and assembly code.

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 prefixes 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.

-  Do *not* use the dedicated or stack registers for other than their intended purpose; 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).

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 `save temporary files` option (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 .s 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` option set, this produces the following:


```
_asmfunc:
    link    4;
    P1.L = .epcbss; P1.H = .epcbss;
    [P1+ 0] = R0;
    [P1+ 4] = R1;
    [P1+ 8] = R2;
    R0 =    12345 (X);
    JUMP    _P1L2147483647;
    JUMP    _P1L2147483647;
_P1L2147483647:
    P0=[FP+ 4];
    unlink;
    JUMP    (P0);
._asmfunc.end
```

Calling C/C++ Functions from Assembly Programs

You may want to call a 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 1-128](#), 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.

 Do *not* use the dedicated registers for other than their intended purpose; the compiler, libraries, debugger, and interrupt routines all depend on having a stack available as defined by those registers.

Preserved registers can be used; their contents will not be changed by calling a C/C++ function. The function will always save and restore 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 (the `-save-temps` command-line switch). By examining the contents of volatile global variables in `*.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, make sure 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

It is necessary to be able to 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.

C/C++ and Assembly Interface

To name an assembly symbol that corresponds to a C/C++ symbol, add an underscore prefix to the C/C++ symbol name when declaring the symbol in assembly. For example, the C/C++ symbol `main` becomes the assembly symbol `_main`.

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 `.GLOBAL` 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 1-20 shows several examples of the C/C++ and assembly interface naming conventions.

Table 1-20. C/C++ Naming Conventions for Symbols

| In the C/C++ Program | In the Assembly Subroutine |
|---|---|
| <code>int c_var; /*declared global*/</code> | <code>.global _c_var; .type _c_var,STT_OBJECT;</code> |
| <code>void c_func(void);</code> | <code>.global _c_func; .type _c_func,STT_FUNC;</code> |
| <code>extern int asm_var;</code> | <code>.global _asm_var; .type _asm_var,STT_OBJECT; .byte = 0x00,0x00,0x00,0x00</code> |
| <code>extern void asm_func(void);</code> | <code>.global _asm_func; .type _asm_func,STT_FUNC; _asm_func:</code> |

C++ Programming Examples

This section shows examples of the features specific to C++. These examples are:

- “Using Fract Support”
- “Using Complex Support”

By default, the compiler runs in C mode. To run the compiler in C++ mode, use the corresponding option on the command line, or select the option in the **Project Options** dialog box in VisualDSP++.

For example, the following command line

```
ccblkfn -c++ source.cpp -21535
```

runs the compiler with

| | |
|------------|---|
| -c++ | Specifies the following source file is written in ANSI/ISO standard C++ extended with the Analog Devices keywords |
| source.cpp | Specifies the source file for your program |
| -21535 | Specifies the Linker Description File for the ADSP-21535 DSP system |

Using Fract Support

[Listing 1-1](#) demonstrates the compiler support for the `fract` type and associated arithmetic operators, such as `+` and `*`. The dot product algorithm is expressed using the standard arithmetic operators. The code demonstrates how two variable length arrays are initialized with fractional literals.

Listing 1-1. Example Code: Using Fract Data Type

```
#include <fract>
#include <stdio.h>
#define N 20
fract x[N] = {.5r,.5r,.5r,.5r,.5r,.5r,.5r,.5r,.5r,
.5r,.5r,.5r,.5r,.5r,.5r,.5r,.5r,.5r,.5r};
fract y[N] = {0,.1r,.2r,.3r,.4r,.5r,.6r,.7r,.8r,.9r,.10r,.1r,
.2r,.3r,.4r,.5r,.6r,.7r,.8r,.9r};
fract fdot(int n, fract *x, fract *y)
{
    int j;
    fract s;
    s = 0;
    for (j=0; j<n; j++)
    {
        s += x[j] * y[j];
    }
    return s;
}
int main(void)
{
    fdot(N,x,y);
}
```

Using Complex Support

The Mandelbrot fractal set is defined by the following iteration on complex numbers.

$$z := z * z + c$$

The c values belong to the set for which the above iteration does not diverge to infinity. The canonical set is defined when z starts from zero.

[Listing 1-2](#) demonstrates the Mandelbrot generator expressed in a simple algorithm using the C++ library `complex` class.

Listing 1-2. Mandelbrot Generator Example — C++ Code

```
#include <complex>
int iterate (complex<double> c, complex<double> z, int max)
{
    int n;
    for (n = 0; n<max && abs(z)<2.0; n++)
    {
        z = z * z + c;
    }
    return (n == max ? 0 : n);
}
```

[Listing 1-3](#) shows a C version of the inner computational function of the Mandelbrot generator extracts performance and programming penalties (compared with the C++ version).

Listing 1-3. Mandelbrot Generator Example — C Code

```
int    iterate (double creal, double cimag,
double zreal, double zimag, int max)
{
    double real, imag;
    int n;
    imag = zimag * zimag;

    for (n = 0; n<max && (real+imag)<4.0; n++)
    {
        zimag = 2.0 * zreal * zimag + cimag;
        zreal = real - imag + creal;
        real = zreal * zreal;
        imag = zimag * zimag;
    }
    return (n == max ? 0 : n);
}
```


2 C/C++ RUN-TIME LIBRARY


The C and C++ run-time libraries are collections of functions, macros, and class templates that you can call from your source programs. 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 your software development by providing code for a variety of common needs.

This chapter contains:

- [“C and C++ Run-Time Library Guide” on page 2-3](#)
It 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 2-20](#)
It tabulates the functions that are defined by ANSI standard header files.
- [“C Run-Time Library Reference” on page 2-23](#)
It 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 of value for DSP programming supplied by Analog Devices. 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 in the current release of the run-time libraries. Chapter 3, “[DSP Run-Time Library](#)”, describes a number of signal processing, vector, matrix, and statistical functions that assist DSP code development.

 For more information on the algorithms on which many of the C library’s math functions are based, see Cody, W. J. 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 C++ library reference information in HTML format is included on the software distribution CD-ROM. To access the reference files from VisualDSP++, use the **Help Topics** command (**Help** menu) and select the **Reference** book icon. From the **Online Manuals** topic, you can open any of the library files. You can also manually access the HTML files using a web browser.

C and C++ Run-Time Library Guide

The C/C++ Run-Time Library contains functions that you can call from your source. This section describes how to use the library and provides information on the following topics:

- [“Calling Library Functions” on page 2-3](#)
- [“Using the Compiler’s Built-In Functions” on page 2-4](#)
- [“Linking Library Functions” on page 2-4](#)
- [“Working with Library Header Files” on page 2-8](#)
- [“Abridged C++ Library Support” on page 2-13](#)

For information on the library’s contents, see [“C Run-Time Library Reference” on page 2-23](#).

Calling Library Functions

Like other functions that you use, library functions should be declared. Declarations are supplied in header files, as described in [“Working with Library Header Files” on page 2-8](#).

To use a C/C++ library function, call the function by name and give the appropriate arguments. The names and arguments for each function appear on the function’s reference page. These reference pages appear in [“C Run-Time Library Reference” on page 2-23](#).



Function names are C/C++ function names. If you call a C or C++ run-time library function from an assembly program, you must use the assembly version of the function name (prefix an underscore to the name). For more information on the naming conventions, see [“C/C++ and Assembly Interface” on page 1-128](#).

You can use the archiver (`elfar`), described in the *VisualDSP++ 3.0 Linker and Utilities Manual for Blackfin DSPs*, 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 it 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 in-line version.

To use built-in functions, include the appropriate headers in your source, or 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.

Linking Library Functions

The C/C++ run-time library is organized as several libraries which are catalogued in [Table 2-1](#). The libraries and startup files are installed within the subdirectory `blackfin\lib` of your VisualDSP++ installation.

Many library files are built twice—once for single-threaded environments, and once for multi-threaded environments. The multi-threaded versions have a `"mt"` suffix.

When you call a run-time 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 (`ADSP-Blackfin.LDF`).



All library files are defined for linking either with ADSP-21532 DSPs or with ADSP-21535 DSPs. Therefore, all files are appropriately marked with `"532"` or `"535"`.

Table 2-1. C and C++ Library Files

| Blackfin\lib Directory | Description |
|--|--|
| bootup532.doj, bootup532mt.doj bootup535.doj, bootup535mt.doj | Jump to "start" symbol in C run-time header for ADSP_21532 or ADSP_21535 |
| crt*532.doj crt*535.doj | C run-time header which sets up system environment before calling <code>main()</code> . There is a single source file (<code>basiccrt.s</code>) that is assembled in different ways to produce all the <code>crt*.doj</code> files. The <code>crt*.doj</code> files have suffices to show which options they support. The <code>crt*.doj</code> files may have various combination of suffices to support different system environments. The <code>crt*.doj</code> files may use the following suffices: c = C++ f = File I/O p = Profiling s = Supervisor mode mt = Multi- threaded. |
| crtn532.doj crtn535.doj | C++ cleanup file used for C++ constructors and destructors. |
| halt532.doj halt532mt.doj halt535.doj halt535mt.doj | Debugging termination code for <code>-flags-link -MDCMDLINE</code> . |
| idle532.doj idle532mt.doj idle535.doj idle535mt.doj | Normal "termination" code that enters IDLE loop after "end" of the application |
| libc532.dlb libc532mt.dlb libc535.dlb libc535mt.dlb | Primary ANSI C run-time library |
| libcpp532.dlb libcpp532mt.dlb libcpp535.dlb libcpp535mt.dlb | Primary ANSI C++ run-time library |

C and C++ Run-Time Library Guide

Table 2-1. C and C++ Library Files (Cont'd)

| Blackfin\lib Directory | Description |
|--|---|
| libcpprt_532.dlb libcpprt_532mt.dlb libcpprt_535.dlb libcpprt_535mt.dlb | C++ run-time support library |
| libdsp532.dlb libdsp535.dlb | DSP library |
| libdspm3res532.dlb libdspm3res535.dlb | DSP run-time library functions that avoid use of M3 register [see Emulator documentation]. Used with -reserve M3 option. |
| libio532.dlb libio532_mt.dlb libio535.dlb libio535_mt.dlb | Host-based I/O facilities, as described in “ stdio.h ” on page 2-11. There are versions for normal and multi-threaded applications. |
| libevent532.dlb libevent532mt.dlb libevent535.dlb libevent535mt.dlb | Interrupt handler support library |
| libm3free532.dlb libm3free532mt.dlb libm3free535.dlb libm3free535mt.dlb | Run-time functions that avoid use of M3 register; used for emulator support |
| libm3res532.dlb libm3res532mt.dlb libm3res535.dlb libm3res535mt.dlb | C/C++ run-time library routines that save context, such as setjmp, longjmp, and exception handlers. These variants avoid the M3 register [see Emulator documentation]. Used with -reserve M3 option. There are versions for normal and multi-threaded applications. |
| libprofile532.dlb libprofile535.dlb | Profile support routines |
| librt532.dlb librt532mt.dlb librt535.dlb librt535mt.dlb | C run-time support library; without File I/O |

Table 2-1. C and C++ Library Files (Cont'd)

| Blackfin\lib Directory | Description |
|--|--|
| librt_fileio532.dlb librt_fileio532mt.dlb librt_fileio535.dlb librt_fileio535mt.dlb | C run-time support library, with File I/O |
| libsftflt532.dlb libsftflt535.dlb | Floating-point emulation routines |
| libsmall532.dlb libsmall532mt.dlb libsmall532.dlb libsmall532mt.dlb | Supervisor mode support routines |
| libvcse21532.dlb libvcse21535.dlb | VCSE utility library for the ADSP-21532 and ADSP-21535 processors, required to build a successful VCSE project. Includes the VCSE interface IID comparator function and streamlined printing routines as well as default contents for the VCSE_MemRequest structure. |
| prfflg0_532.doj prfflg0_535.doj prfflg1_532.doj prfflg1_535.doj prfflg2_532.doj prfflg2_535.doj | Profiling initialization routines as selected by -p, -p1, and -p2 compiler options (see “-p[1 2]” on page 1-35) |
| VDK-CORE-21532.dlb VDK-CORE-21535.dlb | Core kernel functional library (APIs) |
| VDK-e-21532.dlb VDK-e-21535.dlb VDK-i-21532.dlb VDK-i-21535.dlb VDK-n-21532.dlb VDK-n-21535.dlb | Kernel support libraries: e = error checking i = full instrumentation n = no instrumentation |
| src | Directory containing sources for the C/C++ Run-Time library |

If you are not using the default `.LDF` file, then either add the appropriate library/libraries to the LDF used for your project, or use the compiler's `-l` switch to specify the library to be added to the link line.

For example, the switches `-lc` `-lccp_blkfn` and `-lccprt_blkfn` will add the C library `libc.dlb` and the C++ libraries' `ltbccp_blkfn.dlb` and `ltccprt_blkfn.dlb` to the list of libraries to be searched by the linker. For more information on the `.LDF` file, see the *VisualDSP++ 3.0 Linker and Utilities Manual for Blackfin DSPs*.

Working with Library Header Files

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

A list of the header files that are supplied with this release of the Blackfin compiler appears in [Table 2-2](#). You should use a C standard text to augment the information supplied in this chapter.

Table 2-2. Standard C Run-Time Library Header Files

| Header | Purpose | Standard |
|-----------------------|--------------------|----------|
| <code>assert.h</code> | Diagnostics | ANSI |
| <code>ctype.h</code> | Character Handling | ANSI |
| <code>errno.h</code> | Error Handling | ANSI |
| <code>float.h</code> | Floating Point | ANSI |
| <code>limits.h</code> | Limits | ANSI |
| <code>locale.h</code> | Localization | ANSI |
| <code>math.h</code> | Mathematics | ANSI |

Table 2-2. Standard C Run-Time Library Header Files (Cont'd)

| Header | Purpose | Standard |
|----------|----------------------|----------|
| setjmp.h | Non-Local Jumps | ANSI |
| signal.h | Signal Handling | ANSI |
| stdarg.h | Variable Arguments | ANSI |
| stddef.h | Standard Definitions | ANSI |
| stdio.h | Input/Output | ANSI |
| stdlib.h | Standard Library | ANSI |
| string.h | String Handling | ANSI |

This section provides descriptions of the header files contained in the C library. The header files are listed in alphabetical order.

assert.h

Contains the `assert` macro.

ctype.h

Contains functions for character handling, such as `isalpha`, `tolower`, and so forth.

errno.h

Provides access to `errno`. This facility is not, in general, supported by the rest of the library.

float.h

Defines the format of floating-point data types. The `FLT_ROUNDS` macro, defined in the header file, is set to the C run-time environment definition of the rounding mode for `float` variables, which is *round-towards-nearest*.

limits.h

Contains definitions of maximum and minimum values for each C data type other than a floating-point type.

locale.h

Contains definitions for expressing numeric, monetary, time, and other data.

math.h

Includes the floating-point mathematical functions of the C run-time library. The mathematical functions are ANSI standard.

The `math.h` header file contains prototypes for functions used to calculate mathematical properties of single-precision floating-type variables. On the Blackfin DSPs, `double` and `float` are both single-precision floating-point types. Additionally, some functions support a 16-bit fractional data type.

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

Some of the functions in this header file exist as both integer and floating point. The floating-point functions typically have an `f` prefix. Make sure you are using the correct one.



The C language provides for 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 reconverted to floating point for the assignment to `y`.

setjmp.h

Contains `setjmp` and `longjmp` for non-local jumps.

signal.h

Provides function prototypes for the standard ANSI `signal.h` routines. It also includes ANSI-standard signal handling functions of the C library.

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

stdarg.h

Contains definitions needed for functions that accept a variable number of arguments. Callers of such functions must include a prototype.

stddef.h

Contains a few common definitions useful for portable programs, such as `size_t`.

stdio.h

Provides a simple interface with a host environment, which may be a simulator, or a debugger attached to a Blackfin DSP board. The `stdio.h` file support the following functions:

- `printf()` to standard output
- `fopen()` of files on the host
- `fprintf()` to standard output, standard error, or host files opened by `fopen()`

C and C++ Run-Time Library Guide

- `fwrite()` to standard output, standard error, or host files opened by `fopen()`
- `fread()` from host files opened by `fopen()`
- `fclose()` of host files opened by `fopen()`

Standard output and standard error are interpreted as being the simulator's or debugger's console window output.

All I/O operations are channeled through the C function `_primIO()`. The assembly label has two underscores, `__primIO`. The `_primIO()` function accepts no arguments. Instead, it examines the I/O control block at label `_primIOCB`. Without external intervention by a host environment, the `_primIO` routine simply returns, which indicates failure of the request.

When the host environment is providing I/O support, the host places a break point at the start of `_primIO()`. Upon entry to `_primIO()`, the data for the request will reside in a control block at the label `_primIOCB`. The host arranges to intercept control when it enters the `_primIO()` routine, and, after servicing the request, returns control to the calling routine.

See [“File I/O Support” on page 1-94](#) for more information.

stdlib.h

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

string.h

Contains string handling functions, including `strcpy` and `memcpy`.

Abridged C++ Library Support

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

The Abridged Library has two major components: Embedded C++ Library (EC++) and Embedded Standard Template Library (ESTL). The Embedded C++ Library is a conforming implementation of the Embedded C++ Library as specified by the Embedded C++ Technical Committee.

This section lists and briefly describes the following components of the Abridged Library:

- [“Embedded C++ Library Header Files” on page 2-13](#)
- [“C++ Header Files for C Library Facilities” on page 2-15](#)
- [“Embedded Standard Template Library Header Files” on page 2-17](#)

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

Embedded C++ Library Header Files

The following section provides a brief description of the header files in the Embedded C++ Library.

complex

Defines a template that supports the complex class and a set of arithmetic operators.

exception

Defines the `exception` and `bad_exception` classes and several functions for exception handling.

C and C++ Run-Time Library Guide

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.h`.

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.h`.

new

Declares several classes and functions for memory allocations and decollations.

ostream

Defines the `ostream` class for `iostream` insertions.

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.h`.

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. If the name of a C standard library header file is `foo.h`, then the name of the equivalent C++ header file will be `cfoo`. For example, the C++ header file `cstdio` provides the same facilities as the C header file `stdio.h`.

C and C++ Run-Time Library Guide

[Table 2-3](#) lists the C++ header files that provide access to the C library facilities.

Normally, the C standard headers files may be used to define names in the C++ global namespace while the equivalent C++ header files define names in the standard namespace. However, the standard namespace is not supported in this release of the compiler. Therefore, the effect of including one of the C++ header files is the same as including the equivalent C standard library header file.

Table 2-3. C++ Header Files for C Library Facilities

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

Embedded Standard Template Library 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. The Embedded Standard Template Library headers are:

algorithm

Defines numerous common operations on sequences.

deque

Defines a deque template container.

functional

Defines numerous function objects.

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.

C and C++ Run-Time Library Guide

memory

Defines facilities for managing memory.

numeric

Defines several numeric operations on sequences.

queue

Defines two queue template container adapters.

set

Defines two set template containers.

stack

Defines a stack template container adapter.

utility

Defines an assortment of utility templates.

vector

Defines a vector template container.

The Embedded C++ library also includes several headers for compatibility with traditional C++ libraries, such as:

fstream.h

Defines several `istream` template classes that manipulate external files.

iomanip.h

Defines several `istream` 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.

Documented Library Functions

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

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

Table 2-4. Library Functions in the `ctype.h` Header File

| | | |
|-------------------------|--------------------------|-------------------------|
| isalnum | isalpha | isctrl |
| isdigit | isgraph | islower |
| isprint | ispunct | isspace |
| isupper | isxdigit | tolower |
| toupper | | |

Table 2-5. Library Functions in the `math.h` Header File

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

Table 2-6. Library Functions in the `setjmp.h` Header File

| | |
|----------------------|---------------------|
| <code>longjmp</code> | <code>setjmp</code> |
|----------------------|---------------------|

Table 2-7. Library Functions in the `signal.h` Header File

| | | |
|--------------------|---------------------|------------------------|
| <code>raise</code> | <code>signal</code> | <code>interrupt</code> |
|--------------------|---------------------|------------------------|

Table 2-8. Library Functions in the `stdarg.h` Header File

| | | |
|---------------------|---------------------|-----------------------|
| <code>va_arg</code> | <code>va_end</code> | <code>va_start</code> |
|---------------------|---------------------|-----------------------|

Table 2-9. Library Functions in the `stdio.h` Header File

| | | |
|---------------------|----------------------|---------------------|
| <code>fopen</code> | <code>fclose</code> | <code>fread</code> |
| <code>fwrite</code> | <code>fprintf</code> | <code>printf</code> |

Table 2-10. Library Functions in `stdlib.h` Header File

| | | |
|----------------------|----------------------|----------------------|
| <code>abort</code> | <code>abs</code> | <code>atexit</code> |
| <code>atof</code> | <code>atoi</code> | <code>atol</code> |
| <code>bsearch</code> | <code>calloc</code> | <code>div</code> |
| <code>exit</code> | <code>free</code> | <code>labs</code> |
| <code>ldiv</code> | <code>malloc</code> | <code>qsort</code> |
| <code>rand</code> | <code>realloc</code> | <code>srand</code> |
| <code>strtod</code> | <code>strtol</code> | <code>strtoul</code> |

Table 2-11. Library Functions in `string.h` Header File

| | | |
|----------------------|----------------------|-----------------------|
| <code>memchr</code> | <code>memcmp</code> | <code>memcpy</code> |
| <code>memmove</code> | <code>memset</code> | <code>strcat</code> |
| <code>strchr</code> | <code>strcmp</code> | <code>strcoll</code> |
| <code>strcpy</code> | <code>strcspn</code> | <code>strerror</code> |

Documented Library Functions

Table 2-11. Library Functions in `string.h` Header File (Cont'd)

| | | |
|-------------------------|-------------------------|-------------------------|
| strlen | strncat | strncmp |
| strncpy | strpbrk | strrchr |
| strspn | strstr | strtok |
| strxfrm | | |

C Run-Time Library Reference

The C Run-Time Library is a collection of functions called from your C programs. Note 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.

The reference pages for the library functions use the following format:

Name and purpose of the function

Synopsis—Required header file and functional prototype

Description—Function specification

Error Conditions—How the function indicates 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 an error condition.

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

Error Conditions

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

Example

```
#include <stdlib.h>
int i;
i = abs(-5); /* i == 5 */
```

See Also

[fabs](#), [labs](#)

acos

arc cosine

Synopsis

```
#include <math.h>
double acos (double x);
float acosf (float x);
fract16 acos_fr16 (fract16 x);
```

Description

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

The `acos_fr16` function is defined for fractional input values between 0 and 0.9. The input argument is in radians. The output from the function is in radians and is in the range $[\text{acos}(0)*2/\pi, \text{acos}(0.9)*2/\pi]$.

Error Conditions

The `acos` function returns 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](#)

asin

arc sine

Synopsis

```
#include <math.h>
double asin (double x);
float asinf (float x);
fractl6 asin_fr16(fractl6 x);
```

Description

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

The `asin_fr16` function is defined for fractional input values in the range $[-0.9, 0.9]$. The input argument is in radians. The output from the function is in radians and is in the range $[\text{asin}(-0.9)*2/\pi, \text{asin}(0.9)*2/\pi]$.

Error Conditions

The `asin` function returns 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>
double atan (double x);
float atanf (float x);
fract16 atan_fr16 (fract16 x);
```

Description

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

The `atan_fr16` function is defined for fractional input values in the range $[-1.0, 1.0]$. The output from the function is in radians and is in the range $[-\pi/4, \pi/4]$.

Error Conditions

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

Example

```
#include <math.h>
double y;
y = atan(0.0); /* y = 0.0 */
```

See Also

[atan2](#), [tan](#)

atan2

arc tangent of quotient

Synopsis

```
#include <math.h>
double atan2 (double x, double y);
float atan2f (float x, float y);
fract16 atan2_fr16 (fract16 x, fract16 y);
```

Description

The `atan2` function computes the arc tangent of the input value `x` divided by input value `y`. The output, in radians, is in the range $[-\pi/2, \pi/2]$.

Error Conditions

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

Example

```
#include <math.h>
double a;
float b;

a = atan2 (0.0, 0.5); /* the error condition: a = 0.0 */
b = atan2f (1.0, 0.0); /* b = pi/ 2  /
```

See Also

[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 `atexit`.

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 to a double 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). Conversion terminates at the first non-digit (exceptions are “.”, “e”, “E”, and exponents, including the sign).



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

Error Conditions

The `atof` function returns a zero if no conversion can be made.

Example

```
#include <stdlib.h>
double x;

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

See Also

[atoi](#), [atol](#), [strtoul](#), [strtoul](#)

C Run-Time Library Reference

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 can be made.

Example

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

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

See Also

[atol](#), [strtol](#), [atof](#), [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 can be made.

Example

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

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

See Also

[atoi](#), [strtol](#), [strtoul](#), [atof](#)

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 executes a binary search operation on a presorted array, where:

- `key` points to the element to search for
- `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` points to the function used to compare two elements. It takes two parameters—a pointer to the key, and a pointer to an array element. It should return a value less than, equal to, or greater than zero, according to whether the first parameter is less than, equal to, or greater than the second.

The `bsearch` function returns a pointer to the first occurrence of `key` in the array.

Error Conditions

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

Example

```
#include <stdlib.h>
char *answer;
char base[50][3];

answer = bsearch("g", base, 50, 3, strcmp);
```

See Also

[qsort](#)

calloc

allocate and initialize memory

Synopsis

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

Description

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

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>
double ceil(double f);
float ceilf(float f);
```

Description

The `ceil` function returns the smallest integral value that is not less than its input.

Error Conditions

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

Example

```
#include <math.h>
double y;
y = ceil(1.05); /* y = 2.0 */
```

See Also

[floor](#)

cos

cosine

Synopsis

```
#include <math.h>
double cos(double x);
float cosf (float x);
fract16 cos_fr16 (fract16 x)
```

Description

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

The `cos_fr16` function inputs a fractional value in the range $[-1.0, 1.0]$ corresponding to $[-\pi/2, \pi/2]$. The domain represents half a cycle which can be used to derive a full cycle if required. The result, in radians, is in the range $[-1.0, 1.0]$.

Error Conditions

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

Example

```
#include <math.h>
double y;
y = cos(3.14159); /* y = -1.0 */
```

See Also

[acos](#), [sin](#)

cosh

hyperbolic cosine

Synopsis

```
#include <math.h>
double cosh(double x);
float coshf (float x);
```

Description

The `cosh` function returns the hyperbolic cosine of its argument.

Error Conditions

The `cosh` function returns the constant `HUGE_VAL` if the argument is outside the domain.

Example

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

y = cosh(x);
```

See Also

[sinh](#)

C Run-Time Library Reference

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`,

```
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](#), [fmod](#), [modf](#)

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 microprocessor is put into the IDLE state. The `status` argument is stored in register R0, and control is passed to the label `__lib_prog_term`, which is defined in the run-time header.

Error Conditions

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

Example

```
#include <stdlib.h>

exit(EXIT_SUCCESS);
```

See Also

[atexit](#), [abort](#)

exp

exponential

Synopsis

```
#include <math.h>
double exp(double f);
float expf (float f);
```

Description

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

Error Conditions

The `exp` function returns the value `HUGE_VAL` if the argument `f` is greater than the function's domain. The `exp` function returns a zero when the argument is less than its domain.

Example

```
#include <math.h>
double y;
y = exp(1.0); /* y = 2.71828...*/
```

See Also

[pow](#), [log](#)

fabs

float absolute value

Synopsis

```
#include <math.h>
double fabs(double f);
float fabsf(float f);
```

Description

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

Error Conditions

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

Example

```
#include <math.h>
double y;
y = fabs(-2.3); /* y = 2.3 */
y = fabs(2.3);  /* y = 2.3 */
```

See Also

[abs](#), [labs](#)

floor

floor

Synopsis

```
#include <math.h>
double floor(double f);
float floorf (float f);
```

Description

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

Error Conditions

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

Example

```
#include <math.h>
double y;
y = floor(1.25); /* y = 1.0 */
y = floor(-1.25); /* y = -2.0 */
```

See Also

[ceil](#)

fmod

floating-point modulus

Synopsis

```
#include <math.h>
double fmod(double numer, double denom);
float fmodf (float numer, float denom);
```

Description

The `fmod` function computes the floating-point remainder that results from dividing the second argument into the first argument. This value is less than the second argument and has the same sign as the first argument. If the second argument is equal to zero, `fmod` returns a zero.

Error Conditions

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

Example

```
#include <math.h>
double y;
y = fmod(5.0, 2.0); /* y = 1.0 */
```

See Also

[div](#), [modf](#)

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 = malloc(10);    /* Allocate 10 words from heap */
free(ptr);           /* Return space to free heap */
```

See Also

[calloc](#), [malloc](#), [realloc](#)

frexp

separate fraction and exponent

Synopsis

```
#include <math.h>
double frexp(double f, int *exp_ptr);
float frexpf (float f, int *exp_ptr);
```

Description

The `frexp` function separates a floating-point input into a normalized fraction and a (base 2) exponent. The function returns a fraction which is in the interval $[\frac{1}{2}, 1)$, and stores a power of 2 in the integer pointed to by the second argument. If the input is zero, then zeros are stored in both arguments.

Error Conditions

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

Example

```
#include <math.h>
double y;
int exponent;
y = frexp(2.0, &exponent); /* y=0.5, exponent=2 */
```

See Also

[modf](#)

interrupt

define interrupt handling

Synopsis

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

Description

The `interrupt` function determines how a signal received during program execution is handled. The `interrupt` function executes the function pointed to by `func` at every `interrupt sig`; the `signal` function executes the function only once.

The `func` argument must be one of the values listed in [Table 2-12](#). The `interrupt` function causes the receipt of the signal number `sig` to be handled in one of the following ways..

Table 2-12. Interrupt Handling

| Func Value | Action |
|----------------------|---|
| <code>SIG_DFL</code> | The default action is taken. |
| <code>SIG_IGN</code> | The signal is ignored. |
| Function address | The function pointed to by <code>func</code> is executed. |

The function pointed to by `func` is executed each time the interrupt is received. The `interrupt` function must be called with the `SIG_IGN` argument to disable interrupt handling. The `sig` argument may be any of the signals shown in [Table 2-13](#) which lists the supported signals in interrupt priority order from highest to lowest.

See Also

[raise](#), [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 nonzero 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 nonzero value.

Error Conditions

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

Example

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

See Also

[isalnum](#), [islower](#), [isupper](#)

isctrl

detect control character

Synopsis

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

Description

The `isctrl` function determines whether the argument is a control character (0x00-0x1F or 0x7F). If the argument is not a control character, `isctrl` returns a zero. If the argument is a control character, `isctrl` returns a nonzero value.

Error Conditions

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

Example

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

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%2s", isctrl(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 nonzero value.

Error Conditions

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

Example

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

See Also

[isalpha](#), [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 nonzero 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](#)

C Run-Time Library Reference

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 nonzero 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](#)

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 nonzero 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 white space character (0x09-0x0D or 0x20). This includes white 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 nonzero 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

[isctrl](#), [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 nonzero 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 character is a hexadecimal digit character (A-F, a-f, or 0-9). If the argument is not a hexadecimal digit, `isxdigit` returns a zero. If the argument is a hexadecimal digit, `isxdigit` returns a non-zero value.

Error Conditions

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

Example

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

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

See Also

[isalnum](#), [isdigit](#)

labs

long integer absolute value

Synopsis

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

Description

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

Error Conditions

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

Example

```
#include <stdlib.h>
long int j;
j = labs(-285128); /* j = 285128 */
```

See Also

[abs](#), [fabs](#)

ldexp

multiply by power of 2

Synopsis

```
#include <math.h>
double ldexp(double x, int n);
float ldexpf(float x, int n);
```

Description

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

Error Conditions

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

Example

```
#include <math.h>
double y;
y = ldexp(0.5, 2); /* y = 2.0 */
```

See Also

[exp](#), [pow](#)

ldiv

division

Synopsis

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

Description

The `ldiv` function divides `numer` by `denom`, and returns a structure of type `ldiv_t`. The type `ldiv_t` is defined as:

```
typedef struct {
    long int quot;
    long int rem;
} ldiv_t;
```

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

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

Error Conditions

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

Example

```
#include <stdlib.h>
ldiv_t result;

result = ldiv(7, 2);    /* result.quot=3, result.rem=1 */
```

See Also

[div](#), [fmod](#)

log

natural logarithm

Synopsis

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

Description

The `log` function computes the natural (base e) logarithm of its input.

Error Conditions

The `log` function returns `-HUGE_VAL` if the input is zero or negative.

Example

```
#include <math.h>
double y;
y = log(1.0);    /* y = 0.0 */
```

See Also

[exp](#), [log10](#)

log10

base 10 logarithm

Synopsis

```
#include <math.h>
double log10(double f);
float log10f (float f);
```

Description

The `log10` function returns the base 10 logarithm of its input.

Error Conditions

The `log10` function returns `-HUGE_VAL` if the input is zero or negative.

Example

```
#include <math.h>
double y;
y = log10(100.0); /* y = 2.0 */
```

See Also

[log](#), [pow](#)

longjmp

second return from `setjmp`

Synopsis

```
#include <setjmp.h>
void longjmp(jmp_buf env, int return_val);
```

Description

The `longjmp` function causes the program to execute a second return from the place where `setjmp (env.)` was called (in same `env` buffer).

The `longjmp` function takes as its arguments a jump buffer that contains the context at the time of the original `setjmp`. It also takes an integer, `return_val`, which `setjmp` returns if `return_val` is nonzero. 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. Also, automatic variables that are local to the original function calling `setjmp`, that do not have `volatile` qualified type, and that have changed their value prior to the `longjmp` call, have indeterminate value.

Error Conditions

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

Example

```
#include
#include
#include
#include

jmp_buf env;
```

C Run-Time Library Reference

```
int res;

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.

Error Conditions

The `malloc` function returns a null pointer if it is unable to allocate the requested memory.

Example

```
#include <stdlib.h>
int *ptr;

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

See Also

[calloc](#), [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`. It returns a positive lexical value if the `s1` object is lexically greater than the `s2` object, a negative value if the `s2` object is lexically greater than the `s1` object, and a zero if the objects are the same.

Error Conditions

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

Example

```
#include <string.h>
char string1 = "ABC";
char string2 = "BCD";
int result;

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

See Also

[strcmp](#), [strcoll](#), [strncmp](#)

memcpy

copy characters from one object to another

Synopsis

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

Description

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

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="SRC" */
```

See Also

[memmove](#), [strcpy](#), [strncpy](#)

memmove

copy characters from one object to another

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, 5);      /* *ptr = "ABCDE" */
```

See Also

[memcpy](#), [strcpy](#), [strncpy](#)

memset

set range of memory to a character

Synopsis

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

Description

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

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

Error Conditions

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

Example

```
#include <string.h>
char string1[50];
memset(string1, '\0', 50); /* set string1 to 0 */
```

See Also

[memcpy](#)

modf

separate integral and fractional parts

Synopsis

```
#include <math.h>
double modf(double f, double *fraction);
float modff (float f, float *fraction);
```

Description

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

Error Conditions

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

Example

```
#include <math.h>
double y, n;
y = modf(-12.345, &n); /* y = -0.345, n = -12.0 */
```

See Also

[frexp](#)

C Run-Time Library Reference

pow

raise to a power

Synopsis

```
#include <math.h>
double pow(double f, double y);
float powf (float f, float y);
```

Description

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

Error Conditions

The function returns 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, then the function will return the constant `HUGE_VAL`.

Example

```
#include <math.h>
double z;
z = pow(4.0, 2.0); /* z = 16.0 */
```

See Also

[exp](#), [ldexp](#)

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 presorted 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` points to the function used to compare two elements. It takes two parameters—a pointer to the key and a pointer to an array element. The `compare` returns a value less than, equal to, or greater than zero, according to whether the first parameter is less than, equal to, or greater than the second.

C Run-Time Library Reference

Error Condition

The `qsort` function returns no value.

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

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

The `raise()` function sends the signal `sig` to the executing program. The `raise` function forces interrupts wherever possible and simulates an interrupt otherwise. The `sig` argument must be one of the signals listed in priority order in [Table 2-13](#).

Table 2-13. 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 |

Table 2-13. Raise Function Signals — Values and Meanings

| Sig Value | Definition |
|-----------|--------------------|
| SIGBUS | software interrupt |
| SIGFPE | software interrupt |
| SIGSEGV | software interrupt |
| 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 nonzero 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 $[0, 2^{32}-1]$.

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 on the order of $2^{32}-1$.

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 those in the object pointed to by `ptr`. If the size of the new object is greater than the size of the object pointed to by `ptr`, then the values in the newly allocated section are undefined.

If `ptr` is a non-null pointer that was not allocated with `malloc` or `calloc`, the behavior is undefined. If `ptr` is a null pointer, `realloc` imitates `malloc`. If `size` is zero and `ptr` is not a null pointer, `realloc` imitates `free`.

Error Conditions

If memory can not be allocated, `ptr` remains unchanged and `realloc` returns a null pointer.

Example

```
#include <stdlib.h>
int *ptr;
ptr = (int *)malloc(10);           /* intervening code */
ptr = (int *)realloc(ptr, 20);     /* the size is now 20 */
```

See Also

[calloc](#), [free](#), [malloc](#)

setjmp

label for external linkage

Synopsis

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Description

The `setjmp` function is a label declared with external linkage. The `setjmp` label saves the calling environment in the `jmp_buf` argument.

When `setjmp` is called, it immediately returns with zero. The call, in effect, declares the label. If at some later point, `longjmp` is called within the same `jmp_buf` argument, `setjmp` will return a nonzero value. The call to `longjmp` causes a transfer to the label declared with `setjmp`.

Error Conditions

The label `setjmp` does not return an error condition.

Example

See code example for “[longjmp](#)” on [page 2-65](#).

See Also

[longjmp](#)

signal

define signal handling

Synopsis

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

Description

The `signal()` function determines how a signal received during program execution is handled. It causes a single occurrence of an interrupt to be responded to. The `sig` argument must be one of the signals listed in priority order in [Table 2-13](#).



Event handlers may also be installed directly; see [“Interrupt Handler Support” on page 1-99](#). 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_DEL`, the event is enabled, but the dispatcher handles the event and return immediately, without calling a signal handler.

If the function is `SIG_IGN`, the event is disabled.

When operating in user mode, all events are enabled on startup, and left enabled.

See Also

[interrupt](#), [raise](#)

sin

sine

Synopsis

```
#include <math.h>
double sin(double x);
float sinf(float x);
fractl6 sin_fr16(float x);
```

Description

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

The `sin_fr16` function inputs a fractional value in the range $[-1.0, 1.0]$ corresponding to $[-\pi/2, \pi/2]$. The domain represents half a cycle which can be used to derive a full cycle if required. The result, in radians, is in the range $[-1.0, 1.0]$.

Error Conditions

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

Example

```
#include <math.h>
double y;
y = sin(3.14159); /* y = 0.0 */
```

See Also

[asin](#), [cos](#)

sinh

hyperbolic sine

Synopsis

```
#include <math.h>
double sinh(double x);
float sinhf (float x);
```

Description

The `sinh` function returns the hyperbolic sine of x .

Error Conditions

The `sinh` function returns `HUGE_VAL` if the argument is outside the domain.

Example

```
#include <math.h>
double x,y;
y = sinh(x);
```

See Also

[cosh](#)

sqrt

square root

Synopsis

```
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
fract16 sqrt_fr16(fract16 x);
```

Description

The `sqrt` function returns the positive square root of `x`.

Error Conditions

The `sqrt` function returns a zero for a negative input.

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](#)

strcat

concatenate strings

Synopsis

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

Description

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

Error Conditions

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

Example

```
#include <string.h>
char string1[50];
string1[0] = 'A';
string1[1] = 'B';
string1[2] = '\0';
strcat(string1, "CD"); /* new string is "ABCD" */
```

See Also

[strncat](#)

strchr

find first occurrence of character in string

Synopsis

```
#include <string.h>
char *strchr(const char *s1, int c);
```

Description

The `strchr` function returns a pointer to the first location in `s1`, a null-terminated string that contains the character `c`.

Error Conditions

The `strchr` function returns null if `c` is not part of the string.

Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strchr(ptr1, 'E');
/* ptr2 points to the E in TESTING */
```

See Also

[memchr](#), [strrchr](#)

strcmp

compare strings

Synopsis

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

Description

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

Error Conditions

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

Example

```
#include <string.h>
char string1[50], string2[50];
if (strcmp(string1, string2))
    printf("%s is different than %s \n", string1, string2);
```

See Also

[memcmp](#), [strncmp](#)

strcoll

compare strings

Synopsis

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

Description

The `strcoll` function compares the string pointed to by `s1` to 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 DSP's system 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`. 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
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 references to this function.

strlen

string length

Synopsis

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

Description

The `strlen` function returns the length of the null-terminated string pointed to by `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

No references to this function.

strncat

concatenate characters from one string to another

Synopsis

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

Description

The `strncat` function appends a copy of up to `n` characters in the null-terminated string pointed to by `s2` to the end of the null-terminated string pointed to by `s1`. 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 (`'\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`. It 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 have been written.

Error Conditions

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

Example

```
#include <string.h>
char string1[50];
strncpy(string1, "MOREFUN", 4);
/* MORE is copied into string1 */
string1[4] = '\0'; /* must null-terminate string1 */
```

See Also

[memcpy](#), [memmove](#), [strcpy](#)

strpbrk

find character match in two strings

Synopsis

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

Description

The `strpbrk` function returns a pointer to the first character in `s1` that is also found in `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

Error Conditions

In the event that no character in `s1` matches any in `s2`, a null pointer is returned.

Example

```
#include <string.h>
char *ptr1, *ptr2, *ptr3;

ptr1 = "TESTING";
ptr2 = "SHOP"
ptr3 = strpbrk(ptr1, ptr2);
/* ptr3 points to the S in TESTING */
```

See Also

[strcspn](#), [strpbrk](#)

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 portion of string to double representation

Synopsis

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

Description

The `strtod` function returns the value that was represented by the string `nptr` as a double. The `strtod` function expects `nptr` to point to a string of the form:

`[whitespace] [sign] [digits] [.digits] [{d|D|e|E}[sign]digits]`

The white space may consist of space and tab characters, which are ignored; sign is either plus (+) or minus (−); and digits are one or more decimal digits. If no digits appear before the radix character (.), at least one digit must appear after the radix character.

The decimal digits can be followed by an exponent, which consists of an introductory letter (d, D, e, or E) and an optionally signed integer. If neither an exponent part nor a radix character appears, a radix character is assumed to follow the last digit in the string. The first character that does not fit this form stops the scan.

If `endptr` is not NULL, a pointer to the character that stopped the scan is stored at the location pointed to by `endptr`. If no conversion can be performed, the value of `nptr` is stored at the location pointed to by `endptr`.

Error Conditions

The `strtod` function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`. If the correct value results in an overflow, a positive or negative (as appropri-

ate) `HUGE_VAL` is returned. If the correct value results in an underflow, 0 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"*/
```

See Also

[atof](#), [strtoul](#), [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 `s2`.

A call to `strtok`, with `s1` not NULL, returns a pointer to the first token in `s1`, where a token is a consecutive sequence of characters not in `s2`. `s1` is modified in place to insert a null character at the end of the 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 references to this function.

strtol

convert string to long integer

Synopsis

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

Description

The `strtol` function returns as a `long int` the value that was 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 can be made and the invalid string is stored in the object pointed to by `endptr`. 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` 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](#), [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:

- white space (as determined by `isspace`)
- initial characters
- unrecognized characters including a terminating null character

The initial characters may comprise an optional sign character, `0x` or `0X`, when `base` is 16, and those letters and digits which represent an integer with a radix of `base`. The letters (`a-z` or `A-Z`) are assigned the values 10 to 35, and are permitted only when those values are less than the value of `base`.

If `base` is zero, the base is taken from the initial characters. A leading `0x` indicates base 16; a leading `0` indicates base 8. For any other leading characters, base 10 is used. If `base` is between 2 and 36, it is used as a base for conversion.

Error Conditions

The `strtoul` function returns a zero if no conversion can be made and the invalid string is stored in the object pointed to by `endptr`. If the correct value results in an overflow, `ULONG_MAX` is returned. `ERANGE` is stored in `errno` in the case of overflow.

Example

```
#include <stdlib.h>
#define base 10

char *rem;
unsigned long int i;

i = strtoul("2345.5", &rem, base);
/* i = 2345, rem = "5" */
```

See Also

[atoi](#), [atol](#), [strtol](#)

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>
double tan(double x);
float tanf(float x);
fract16 tan_fr16(fract16 x);
```

Description

The `tan` function returns the tangent of the argument. The input, in radians, must be in the range $[-9099, 9099]$.

The `tan_fr16` function is defined for fractional input values between $[-\pi/4, \pi/4]$. The result is in radians in the range $[-1.0, 1.0]$.

Error Conditions

The `tan` function returns zero if the input argument is outside 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>
double tanh(double x);
float tanhf (float x);
```

Description

The `tanh` function returns the hyperbolic tangent of x .

Error Conditions

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

Example

```
#include <math.h>
double x,y;
y = tanh(x);
```

See Also

[sinh](#), [cosh](#)

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](#)

C Run-Time Library Reference

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

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

Error Conditions

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

Example

```
#include <stdarg.h>
#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 = '';
va_start (ap,s1);
s = s1;
while (s){
    strcat (result,s);
    s = va_arg (ap,char *);
}
va_end (ap);
return result;
}
```

See Also

[va_start](#), [va_end](#)

va_end

finish variable-length argument list processing

Synopsis

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

Description

The `va_end` macro can only be 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.

Example

See “[va_arg](#)” on page 2-116.

See Also

[va_arg](#), [va_start](#)

va_start

initialize the variable-length argument list processing

Synopsis

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

Description

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

Error Conditions

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

Example

See [“va_arg” on page 2-116](#).

See Also

[va_arg](#), [va_end](#)

3 DSP RUN-TIME LIBRARY

This chapter describes the DSP run-time library which contains a broad collection of functions that are commonly required by DSP applications. The services provided by the library include support for general purpose signal processing such as companders, filters, and FFT functions. All these services are Analog Devices extensions to ANSI standard C.

For more information on the algorithms on which many of the C library's math functions are based, see Cody, W. J. and W. Waite, *Software Manual for the Elementary Functions*, Englewood Cliffs, New Jersey: Prentice Hall, 1980.

This chapter contains:

- [“DSP Run-Time Library Guide” on page 3-2](#)
It 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 3-25](#)
It provides 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 you can call from your source program. This section describes how to use the library and provides information about:

- [“Linking DSP Library Functions”](#)
- [“Working With Library Source Code”](#)
- [“DSP Header Files” on page 3-3](#)

Linking DSP Library Functions

When C/C++ code calls a DSP run-time library function, the call creates a reference that the linker resolves when linking your program. This requires the linker to be directed to link with the DSP run-time library, `libdsp.dlb`, in the subdirectory `Blackfin\lib` under the VisualDSP++ installation directory. This is done automatically when you use the default Linker Description File (`.LDF`) for Blackfin DSPs, as this specifies that `libdsp.dlb` will be on each link line. If you do not use the default `.LDF` file, then either add `libdsp.dlb` to the `.LDF` file used by your project, or alternatively use the compiler `-ldsp` switch to specify that `libdsp.dlb` is to be added to the link line.

Working With Library Source Code

The source code for some functions and macros in the DSP run-time library is provided with your VisualDSP++ software. By default, the libraries are installed in the directory `Blackfin\lib` and the source files are copied into `Blackfin\lib\src`. Each function is kept in a separate file. The file name is the name of the function with the extension `.asm` or `.c`. If you do not intend to modify any of the runtime library functions, you can delete this directory and its contents to conserve disk space.

The source code is provided to customize specific functions for your own needs. To modify these files, proficiency in Blackfin assembly language and an understanding of the run-time environment is needed (as explained in [“C/C++ Run-Time Model and Environment” on page 1-113](#)).

Before making any modifications to the source code, copy the source code 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 provided.

DSP Header Files

The DSP header files contains prototypes for all the DSP library functions. When the appropriate `#include` preprocessor command is included in your source, the compiler will use the prototypes to check that each function is called with the correct arguments. The DSP header files included in this release of the `ccblkfn` compiler are:

- [“complex.h — Basic Complex Arithmetic Functions” on page 3-4](#)
- [“filter.h — DSP Filters and Transformations” on page 3-4](#)
- [“math.h — Math Functions” on page 3-10](#)
- [“matrix.h — Matrix Functions” on page 3-12](#)
- [“stats.h — Statistical Functions” on page 3-16](#)
- [“vector.h — Vector Functions” on page 3-18](#)
- [“window.h — Window Generators” on page 3-18](#)

complex.h — Basic Complex Arithmetic Functions

The `complex.h` header file contains type definitions and basic arithmetic operations for variables of type `complex_float`, `complex_double`, and `complex_fract16`.

The complex functions defined in this header file are listed in [Table 3-1](#). The following structures are used to represent complex numbers in rectangular coordinates:

```
typedef struct
{
    float re;
    float im;
} complex_float;

typedef struct
{
    double re;
    double im;
} complex_double;

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

Details of the basic complex arithmetic functions are included in “[DSP Run-Time Library Reference](#)” on page 3-25.

filter.h — DSP Filters and Transformations

The `filter.h` header file contains filters used in digital signal processing. It also includes the A-law and μ -law companders that are used by voice-band compression and expansion applications.

The header file also contains functions that perform key transformations used in DSPs, including FFT and `convolve`.

Table 3-1. Complex Functions

| Description | Prototype |
|-------------------------------|---|
| Complex Absolute Value | double cabs (complex_double a) float cabsf (complex_float a) fract16 cabs_fr16 (complex_fract16 a) |
| Complex Addition | complex_double cadd (complex_double a, complex_double b) complex_float caddf (complex_float a, complex_float b) complex_fract16 cadd_fr16 (complex_fract16 a, complex_fract16 b) |
| Complex Subtraction | complex_double csub (complex_double a, complex_double b) complex_float csubf (complex_float a, complex_float b) complex_fract16 csub_fr16 (complex_fract16 a, complex_fract16 b) |
| Complex Multiply | complex_double cmlt (complex_double a, complex_double b) complex_float cmltf (complex_float a, complex_float b) complex_fract16 cmlt_fr16 (complex_fract16 a, complex_fract16 b) |
| Complex Division | complex_double cdiv (complex_double a, complex_double b) complex_float cdivf (complex_float a, complex_float b) complex_fract16 cdiv_fr16 (complex_fract16 a, complex_fract16 b) |
| Get Phase of a Complex Number | double arg (complex_double a) float argf (complex_float a) fract16 arg_fr16 (complex_fr16 a) |
| Complex Conjugate | complex_double conj (complex_double a) complex_float conjf (complex_float a) complex_fract16 conj_fr16 (complex_fract16 a) |

Table 3-1. Complex Functions (Cont'd)

| Description | Prototype |
|---------------------------|--|
| Complex Polar Coordinates | <code>complex_double polar (double mag, double phase)</code> <code>complex_float polarf (float mag, float phase)</code> <code>complex_fract16 polar_fr16 (fract16 mag, fract16 phase)</code> |
| Complex Exponential | <code>complex_double cexp (double a)</code> <code>complex_float cexpf (float a)</code> |
| Normalization | <code>complex_double norm (complex_double a)</code> <code>complex_float normf (complex_float a)</code> |

Various forms of the FFT function are provided by the library corresponding to radix-2, radix-4, and two dimensional FFTs. The number of points is provided as an argument. The twiddle table for the FFT functions is supplied as a separate argument and is normally calculated once during program initialization.

Library functions are provided to initialize a twiddle table. 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 argument is set to 1, the FFT function will use all the table; if FFT uses only half the number of points of the largest, the stride should be 2.

The functions defined in this header file are listed in [Table 3-2 on page 3-7](#) and [Table 3-3 on page 3-7](#) and are described in detail in “DSP Run-Time Library Reference” on [page 3-25](#).

Table 3-2. Filter Library

| Description | Prototype |
|--|---|
| Finite Impulse Response Filter | void fir_fr16 (const fract16 x[], fract16 y[], int n, fir_state_fr16 *s) |
| Infinite Impulse Response Filter | void iir_fr16 (const fract16 x[], fract16 y[], int n, iir_state_fr16 *s) |
| Fir Decimation Filter | void fir_decima_fr16 (const fract16 x[], fract16 y[], int n, fir_state_fr16 *s) |
| Fir Interpolation Filter | void fir_interp_fr16 (const fract16 x[], fract16 y[], int n, fir_state_fr16 *s) |
| Complex Finite Impulse Response Filter | void cfir_fr16 (const complex_fract16 x[], complex_fract16 y[], int n, cfir_state_fr16 *s) |

Table 3-3. Transformational Functions

| Description | Prototype |
|--|---|
| Fast Fourier Transforms | |
| Generate FFT Twiddle Factors | void twidfft_fr16 (complex_fract16 w[], int n) |
| Generate FFT Twiddle Factors for Radix 2 FFT | void twidfftrad2_fr16 (complex_fract16 w[], int n) |
| Generate FFT Twiddle Factors for Radix 4 FFT | void twidfftrad4_fr16 (complex_fract16 w[], int n) |
| Generate FFT Twiddle Factors for 2-D FFT | void twidfft2d_fr16 (complex_fract16 w[], int n) |

Table 3-3. Transformational Functions (Cont'd)

| Description | Prototype |
|--------------------------------------|--|
| N Point Radix 2 Complex Input FFT | <pre>void cfft_fr16 (const complex_fract16 *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int wst, int n, int block_exponent, int scale_method)</pre> |
| N Point Radix 2 Real Input FFT | <pre>void rfft_fr16 (const fract16 *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int wst, int n, int block_exponent, int scale_method)</pre> |
| N Point Radix 2 Inverse FFT | <pre>void ifft_fr16 (const complex_fract *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int wst, int n, int block_exponent, int scale_method)</pre> |
| N Point Radix 4 Complex Input FFT | <pre>void cfftrad4_fr16 (const complex_fract16 *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int wst, int n, int block_exponent, int scale_method)</pre> |
| N Point Radix 4 Real Input FFT | <pre>void rfftrad4_fr16 (const fract16 *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int wst, int n, int block_exponent, int scale_method)</pre> |
| N Point Radix 4 Inverse Input FFT | <pre>void iffftrad4_fr16 (const complex_fract *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int wst, int n, int block_exponent, int scale_method)</pre> |
| Nxn Point 2-D Complex Input FFT | <pre>void cfft2d_fr16 (const complex_fract16 *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int wst, int n, int block_exponent, int scale_method)</pre> |

Table 3-3. Transformational Functions (Cont'd)

| Description | Prototype |
|---------------------------------|--|
| Nxn Point 2-D Real Input FFT | void rfft2d_fr16 (const fract16 *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int wst, int n, int block_exponent, int scale_method) |
| Nxn Point 2-D Inverse FFT | void ifft2d_fr16 (const complex_fract *in, complex_fract16 *t, complex_fract16 *out, const complex_fract16 *w, int wst, int n, int block_exponent, int scale_method) |
| Convolutions | |
| Convolution | void convolve_fr16 (const fract16 cin1[], int clen1, const fract16 cin2[], int clen2, fract16 cout[]) |
| 2-D Convolution | void conv2d_fr16 (const fract16 *cin1, int crow1, int ccol1, const fract16 *cin2, int crow2, int ccol2, fract16 *cout) |
| 2-D Convolution 3x3 Matrix | void conv2d3x3_fr16 (const fract16 *cin, int crow1, int ccol1, const fract16 cin2 [3] [3], fract16 *cout) |
| Compression/Expansion | |
| A-law compression | void a_compress (const short in[], short out[], int n) |
| A-law expansion | void a_expand (const short in[], short out[], int n) |
| μ -law compression | void mu_compress (const short in[], short out[], int n) |
| μ -law expansion | void mu_expand (const char in[], short out[], int n) |

math.h — Math Functions

The standard math functions have been augmented by implementations for the `float` data type, and in some cases, for the `fract16` data type.

[Table 3-4](#) provides a summary of the functions defined by the `math.h` header file. Descriptions of these functions are given under the name of the `double` version in the “[C Run-Time Library Reference](#)” on [page 2-23](#).

This header file also provides prototypes for a number of additional math function—`clip`, `copysign`, `max`, and `min`, and an integer function, `countones`. These functions are described in the “[DSP Run-Time Library Reference](#)” on [page 3-25](#).

Table 3-4. Math Library

| Description | Prototype |
|-------------------------|--|
| Arc Cosine | <code>double acos (double x)</code> <code>float acosf (float x)</code> <code>fract16 acos_fr16 (fract16 x)</code> |
| Arc Sine | <code>double asin (double x)</code> <code>float asinf (float x)</code> <code>fract16 asin_fr16 (fract16 x)</code> |
| Arc Tangent | <code>double atan (double x)</code> <code>float atanf (float x)</code> <code>fract16 atan_fr16 (fract16 x)</code> |
| Arc Tangent of Quotient | <code>double atan2 (double x, double y)</code> <code>float atan2f (float x, float y)</code> <code>fract16 atan2_fr16 (fract16 x, fract16 y)</code> |
| Ceiling | <code>double ceil (double x)</code> <code>float ceilf (float x)</code> |
| Cosine | <code>double cos (double x)</code> <code>float cosf (float x)</code> <code>fract16 cos_fr16 (fract16 x)</code> |
| Cotangent | <code>double cot (double x)</code> <code>float cotf (float x)</code> |

Table 3-4. Math Library (Cont'd)

| Description | Prototype |
|---------------------------|--|
| Hyperbolic Cosine | double cosh (double x) float coshf (float x) |
| Exponential | double exp (double x) float expf (float x) |
| Floor | double floor (double x) final floorf (float x) |
| Floating-Point Remainder | double fmod (double x, double y) float fmodf (float x, float y) |
| Get Mantissa and Exponent | double frexp (double x, int *n) float frexpf (float x, int *n) |
| Multiply by Power of 2 | double ldexp(double x, int n) float ldexpf(float x, int n) |
| Natural Logarithm | double log (double x) float logf (float x) |
| Logarithm Base 10 | double log10 (double x) float log10f (float x) |
| Get Fraction and Integer | double modf (double x, double *i) float modff (float x, float *i) |
| Power | double pow (double x, double y) float powf (float x, float y) |
| Reciprocal Square Root | double rsqrt(double x) float rsqrtf(float x) |
| Sine | double sin (double x) float sinf (float x) fract16 sin_fr16 (fract16 x) |
| Hyperbolic Sine | double sinh (double x) float sinhf (float x) |
| Square Root | double sqrt (double x) float sqrtf (float x) fract16 sqrt_fr16 (fract16 x) |

Table 3-4. Math Library (Cont'd)

| Description | Prototype |
|--------------------|---|
| Tangent | double tan (double x) float tanf (float x) fract16 tan_fr16 (fract16 x) |
| Hyperbolic Tangent | double tanh (double x) float tanhf (float x) |

matrix.h — Matrix Functions

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 — Basic Complex Arithmetic Functions” on page 3-4](#) for definitions of the complex types.

The matrix functions defined in the `matrix.h` header file are listed in [Table 3-5 on page 3-13](#). In most of the function prototypes:

- *a is a pointer to input matrix a [] []
- *b is a pointer to input matrix b [] []
- b is an input scalar
- n is the number of rows
- m is the number of columns
- *c is a pointer to output matrix c [] []

In the `matrix*matrix` functions, `n` and `k` are the dimensions of matrix `a` and `k` and `m` 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.

Table 3-5. Matrix Functions

| Description | Prototype |
|--|--|
| Real Matrix + Scalar Addition | <pre>void matsadd (const double *a, double b, int n, int m, double *c) void matsaddf (const float *a, float b, int n, int m, float *c) void matsadd_fr16 (const fract16 *a, fract16 b, int n, int m, fract16 *c)</pre> |
| Real Matrix - Scalar Subtraction | <pre>void matssub (const double *a, double b, int n, int m, double *c) void matssubf (const float *a, float b, int n, int m, float *c) void matssub_fr16 (const fract16 *a, fract16 b, int n, int m, fract16 *c)</pre> |
| Real Matrix * Scalar Multiplication | <pre>void matsmlt (const double *a, double b, int n, int m, double *c) void matsmltf (const float *a, float b, int n, int m, float *c) void matsmlt_fr16 (const fract16 *a, fract16 b, int n, int m, fract16 *c)</pre> |
| Real Matrix + Matrix Addition | <pre>void matmadd (const double *a, const double *b, int n, int m, double *c) void matmaddf (const float *a, const float *b, int n, int m, float *c) void matmadd_fr16 (const fract16 *a, const fract16 *b, int n, int m, fract16 *c)</pre> |

Table 3-5. Matrix Functions (Cont'd)

| Description | Prototype |
|--|---|
| Real Matrix – Matrix Subtraction | <pre>void matmsub (const double *a, const double *b, int n, int m, double *c) void matmsubf (const float *a, const float *b, int n, int m, float *c) void matmsub_fr16 (const fract16 *a, const fract16 *b, int n, int m, fract16 *c)</pre> |
| Real Matrix * Matrix Multiplication | <pre>void matmmlt (const double *a, int n, int k, const double *b, int m, double *c) void matmmltf (const float *a, int n, int k, const float *b, int m, float *c) void matmmlt_fr16 (const fract16 *a, int n, int k, const fract16 *b, int m, fract16 *c)</pre> |
| Complex Matrix + Scalar Addition | <pre>void cmatsadd (const complex_double *a, complex_double b, int n, int m, complex_double *c) void cmatsaddf (const complex_float *a, complex_float b, int n, int m, complex_float *c) void cmatsadd_fr16 (const complex_fract16 *a, complex_fract16 b, int n, int m, complex_fract16 *c)</pre> |

Table 3-5. Matrix Functions (Cont'd)

| Description | Prototype |
|---|--|
| Complex Matrix – Scalar Subtraction | <pre> void cmatssub (const complex_double *a, complex_double b, int n, int m, complex_double *c) void cmatssubf (const complex_float *a, complex_float b, int n, int m, complex_float *c) void cmatssub_fr16 (const complex_fract16 *a, complex_fract16 b, int n, int m, complex_fract16 *c) </pre> |
| Complex Matrix * Scalar Multiplication | <pre> void cmatsmlt (const complex_double *a, complex_double b, int n, int m, complex_double *c) void cmatsmltf (const complex_float *a, complex_float b, int n, int m, complex_float *c) void cmatsmlt_fr16 (const complex_fract16 *a, complex_fract16 b, int n, int m, complex_fract16 *c) </pre> |
| Complex Matrix + Matrix Addition | <pre> void cmatmadd (const complex_double *a, const complex_double *b, int n, int m, complex_double *c) void cmatmaddf (const complex_float *a, const complex_float *b, int n, int m, complex_float *c) void cmatmadd_fr16 (const complex_fract16 *a, const complex_fract16 *b, int n, int m, complex_fract16 *c) </pre> |

Table 3-5. Matrix Functions (Cont'd)

| Description | Prototype |
|---|---|
| Complex Matrix – Matrix Subtraction | <pre>void cmatmsub (const complex_double *a, const complex_double *b, int n, int m, complex_double *c) void cmatmsubf (const complex_float *a, const complex_float *b, int n, int m, complex_float *c) void cmatmsub_fr16 (const complex_fract16 *a, const complex_fract16 *b, int n, int m, complex_fract16 *c)</pre> |
| Complex Matrix * Matrix Multiplication | <pre>void cmatmmlt (const complex_double *a, int n, int k, const complex_double *b, int m, complex_double *c) void cmatmmltf (const complex_float *a, int n, int k, const complex_float *b, int m, complex_float *c) void cmatmmlt_fr16 (const complex_fract16 *a, int n, int k, const complex_fract16 *b, int m, complex_fract16 *c)</pre> |
| Transpose | <pre>void transpm (const double *a, int n, int m, double *c) void transpmf (const float *a, int n, int m, float *c) void transpm_fr16 (const fract16 *a, int n, int m, fract16 *c)</pre> |

stats.h — Statistical Functions

The statistical functions defined in the `stats.h` header file are listed in [Table 3-6](#) and are described in detail in “[DSP Run-Time Library Reference](#)” on page 3-25.

Table 3-6. Statistical Functions

| Description | Prototype |
|---------------------|--|
| Autocoherence | <pre>void autocohf (const float a[], int n, int m, float c[]) void autocoh_fr16 (const fract16 a[], int n, int m, fract16 c[])</pre> |
| Autocorrelation | <pre>void autocorrf (const float a[], int n, int m, float c[]) void autocorr_fr16 (const fract16 a[], int n, int m, fract16 c[])</pre> |
| Cross-coherence | <pre>void crosscohf (const float a[], const float b[], int n, int m, float c[]) void crosscoh_fr16 (const fract16 a[], const fract16 b[], int n, int m, fract16 c[])</pre> |
| Cross-correlation | <pre>void crosscorrf (const float a[], const float b[], int n, int m, float c[]) void crosscorr_fr16 (const fract16 a[], const fract16 b[], int n, int m, fract16 c[])</pre> |
| Histogram | <pre>void histogramf (const float a[], int c[], float max, float min, int n, int m) void histogram_fr16 (const fract16 a[], int c[], fract16 max, fract16 min, int n, int m)</pre> |
| Mean | <pre>float meanf (const float a[], int n) fract16 mean_fr16 (const fract16 a[], int n)</pre> |
| Root Mean Square | <pre>float rmsf (const float a[], int n) fract16 rms_fr16 (const fract16 a[], int n)</pre> |
| Variance | <pre>float varf (const float a[], int n) fract16 var_fr16 (const fract16 a[], int n)</pre> |
| Count Zero Crossing | <pre>float zero_crossf (const float a[], int n) fract16 zero_cross_fr16 (const fract16 a[], int n)</pre> |

vector.h — Vector Functions

The `vector.h` header file contains functions for operating on real and complex vectors, both vector-scalar and vector-vector operations. See “[complex.h — Basic Complex Arithmetic Functions](#)” on page 3-4 for definitions of the complex types.

The functions in the `vector.h` header file are listed in [Table 3-7 on page 3-19](#). In the **Prototype** column, `a[]` and `b[]` are input vectors, `b` is an input scalar, `c[]` is an output vector and `n` 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 if the input vectors and the output vector are in different memory banks. This structure avoids any potential memory bank collisions.

window.h — Window Generators

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 3-8](#) and are described in detail in “[DSP Run-Time Library Reference](#)” on page 3-25.

For all window functions, a stride parameter `a` can be used to space the window values. The window length parameter `n` equates to the number of elements in the window. Therefore, for a stride `a` of 2 and a length `n` of 10, an array of length 20 is required, where every second entry is untouched.

Table 3-7. Vector Functions

| Description | Prototype |
|--|--|
| Real Vector + Scalar Addition | <pre>void vecsadd (const double a[], double b, double c[], int n) void vecsaddf (const float a[], float b, float c[], int n) void vecsadd_fr16 (const fract16 a[], fract16 b, fract16 c[], int n)</pre> |
| Real Vector – Scalar Subtraction | <pre>void vecssub (const double a[], double b, double c[], int n) void vecssubf (const float a[], float b, float c[], int n) void vecssub_fr16 (const fract16 a[], fract16 b, fract16 c[], int n)</pre> |
| Real Vector * Scalar Multiplication | <pre>void vecsmlt (const double a[], double b, double c[], int n) void vecsmltf (const float a[], float b, float c[], int n) void vecsmlt_fr16 (const fract16 a[], fract16 b, fract16 c[], int n)</pre> |
| Real Vector + Vector Addition | <pre>void vecvadd (const double a[], const double b [], double c[], int n) void vecvaddf (const float a[], const float b [], float c[], int n) void vecvadd_fr16 (const fract16 a[], const fract16 b [], fract16 c[], int n)</pre> |

Table 3-7. Vector Functions (Cont'd)

| Description | Prototype |
|--|--|
| Real Vector – Vector Subtraction | <pre>void vecvsub (const double a[], const double b [], double c[], int n) void vecvsubf (const float a[], const float b [], float c[], int n) void vecvsub_fr16 (const fract16 a[], const fract16 b [], fract16 c[], int n)</pre> |
| Real Vector * Vector Multiplication | <pre>void vecvmlt (const double a[], const double b [], double c[], int n) void vecvmltf (const float a[], const float b [], float c[], int n) void vecvmlt_fr16 (const fract16 a[], const fract16 b [], fract16 c[], int n)</pre> |
| Maximum Value of Vector Elements | <pre>double vecmax (const double a[], int n) float vecmaxf (const float a[], int n) fract16 vecmax_fr16 (const fract16 a[], int n)</pre> |
| Minimum Value of Vector Elements | <pre>double vecmin (const double a[], int n) float vecminf (const float a[], int n) fract16 vecmin_fr16 (const fract16 a[], int n)</pre> |
| Index of Maximum Value of Vector Elements | <pre>int vecmaxloc (const double a[], int n) int vecmaxlocf (const float a[], int n) fract16 vecmaxloc_fr16 (const fract16 a[], int n)</pre> |
| Index of Minimum Value of Vector Elements | <pre>int vecminloc (const double a[], int n) int vecminlocf (const float a[], int n) fract16 vecminloc_fr16 (const fract16 a[], int n)</pre> |

Table 3-7. Vector Functions (Cont'd)

| Description | Prototype |
|---|--|
| Complex Vector + Scalar Addition | <pre> void cvecsadd (const complex_double a[], complex_double b, complex_double c[], int n) void cvecsaddf (const complex_float a[], complex_float b, complex_float c[], int n) void cvecsadd_fr16 (const complex_fract16 a[], complex_fract16 b, complex_fract16 c[], int n) </pre> |
| Complex Vector – Scalar Subtraction | <pre> void cvecssub (const complex_double a[], complex_double b, complex_double c[], int n) void cvecssubf (const complex_float a[], complex_float b, complex_float c[], int n) void cvecssub_fr16 (const complex_fract16 a[], complex_fract16 b, complex_fract16 c[], int n) </pre> |
| Complex Vector * Scalar Multiplication | <pre> void cvecsmult((const complex_double a[], complex_double b, complex_double c[], int n) void cvecsmultf (const complex_float a[], complex_float b, complex_float c[], int n) void cvecsmult_fr16 (const complex_fract16 a[], complex_fract16 b, complex_fract16 c[], int n) </pre> |

Table 3-7. Vector Functions (Cont'd)

| Description | Prototype |
|--|--|
| Complex Vector + Vector Addition | <pre>void cvecvadd (const complex_double a[], const complex_double b[], complex_double c[], int n) void cvecvaddf (const complex_float a[], const complex_float b[], complex_float c[], int n) void cvecvadd_fr16 (const complex_fract16 a[], const complex_fract16 b[], complex_fract16 c[], int n)</pre> |
| Complex Vector – Vector Subtraction | <pre>void cvecvsub (const complex_double a[], const complex_double b[], complex_double c[], int n) void cvecvsubf (const complex_float a[], const complex_float b[], complex_float c[], int n) void cvecvsub_fr16 (const complex_fract16 a[], const complex_fract16 b[], complex_fract16 c[], int n)</pre> |
| Complex Vector * Vector Mutliplication | <pre>void cvecvmlt (const complex_double a[], const complex_double b[], complex_double c[], int n) void cvecvmltf (const complex_float a[], const complex_float b[], complex_float c[], int n) void cvecvmlt_fr16 (const complex_fract16 a[], const complex_fract16 b[], complex_fract16 c[], int n)</pre> |

Table 3-7. Vector Functions (Cont'd)

| Description | Prototype |
|----------------------------|--|
| Real Vector Dot Product | <pre>double vecdot (const double a[], const double b[], int n) float vecdotf (const float a[], const float b[], int n) fract16 vecdot_fr16 (const fract16 a[], const fract16 b[], int n)</pre> |
| Complex Vector Dot Product | <pre>complex_double cvecdot (const complex_double a[], const complex_double b[], int n) complex_float cvecdotf (const complex_float a[], const complex_float b[], int n) complex_fract16 cvecdot_fr16 (const complex_fract16 a[], const complex_fract16 b[], complex_fract16 c [], int n)</pre> |

Table 3-8. Window Generator Functions

| Description | Prototype |
|--------------------------|--|
| Generate Bartlett Window | <pre>void gen_bartlett_fr16 (fract16 w[], int a, int n)</pre> |
| Generate Blackman Window | <pre>void gen_blackman_fr16 (fract16 w[], int a, int n)</pre> |
| Generate Gaussian Window | <pre>void gen_gaussian_fr16 (fract16 w[], float alpha, int a, int n)</pre> |
| Generate Hamming Window | <pre>void gen_hamming_fr16 (fract16 w[], int a, int n)</pre> |
| Generate Hanning Window | <pre>void gen_hanning_fr16 (fract16 w[], int a, int n)</pre> |

Table 3-8. Window Generator Functions (Cont'd)

| Description | Prototype |
|-----------------------------|---|
| Generate Harris Window | <code>void gen_harris_fr16 (fract16 w[], int a, int n)</code> |
| Generate Kaiser Window | <code>void gen_kaiser_fr16 (fract16 w[], float beta, int a, int n)</code> |
| Generate Rectangular Window | <code>void gen_rectangular_fr16 (fract16 w[], int a, int n)</code> |
| Generate Triangle Window | <code>void gen_triangle_fr16 (fract16 w[], int a, int n)</code> |
| Generate Vonhann Window | <code>void gen_vonhann_fr16 (fract16 w[], int a, int n)</code> |

DSP Run-Time Library Reference

This section provides descriptions of the DSP run-time library functions. The reference pages for the library functions use the following format.

Notation Conventions

The reference pages for the library functions use the following format.

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`, or `fract16`), several prototypes are given.

Description — Function specification

Algorithm — High level mathematical representation of the function

Domain — Range of values supported by function

Notes — Other miscellaneous notations



For some functions, the interface is presented using the “K&R” style for ease of documentation. An ANSI C prototype is provided in the header file.

a_compress

A-law compression

Synopsis

```
#include <filter.h>
void a_compress(in, out, n)
const short in[]; /* Input array */
short out[];      /* Output array */
int n;            /* Number of elements to be compressed */
```

Description

This function performs A-law compression (ITU rec. G.711) on the elements of the input vector pointed to by `in` and outputs compressed data in the vector pointed to by `out`.

Algorithm

$C(k) = \text{a-law compression of } A(k) \text{ for } k=0 \text{ to } n-1$

Domain

content of input array: -4096 to 4095

a_expand

A-law expansion

Synopsis

```
#include <filter.h>
void a_expand(in, out, n)
const short in[]; /* Input array */
short out[];      /* Output array */
int n;            /* Number of elements to be expanded */
```

Description

This function performs A-law expansion (ITU rec. G.711) on the elements of the input vector pointed to by *in* and outputs expanded data in the vector pointed to by *out*.

Algorithm

$C(k)$ = a-law expansion of $A(k)$ for $k = 0$ to $n-1$

Domain

content of input array: 0 to 255

arg

get phase of a complex number

Synopsis

```
#include <complex.h>
float argf (complex_float a);
double arg (complex_double a);
fract16 arg_fr16 (complex_fract16 a);
```

Description

This function computes the phase of complex input *a* and returns the result.

Algorithm

$$c = \text{atan}\left(\frac{\text{Im}(a)}{\text{Re}(a)}\right)$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for `argf ()`, `arg ()`

-1.0 to $+1.0$ for `arg_fr16 ()`

Note

$\text{Im}(a) / \text{Re}(a) \leq 1$ for `arg_fr16 ()`

autocoh

autocoherence

Synopsis

```

#include <stats.h>
void autocohf(a,n,m,c)
const float a[];    /* Input vector a */
int n;              /* Input samples */
int m;              /* Lag count */
float c[];          /* Output vector c */
void autocoh_fr16 (a,n,m,c)
const fract16 a[]; /* Input vector a */
int n;             /* Input samples */
int m;             /* Lag count */
fract16 c[];       /* Output vector c */

```

Description

This function computes the autocohereence of the input elements contained within input vector *a*, and stores the result to output vector *c*.

Algorithm

$$c_k = \frac{1}{n} * \left(\sum_{j=0}^{n-k-1} (a_j - \bar{a}) * (a_{j+k} - \bar{a}) \right)$$

where $k = \{0,1,...,m-1\}$ and \bar{a} is the mean value of input vector *a*.

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for autocohf ()

-1.0 to 1.0 for autocoh_fr16 ()

autocorr

autocorrelation

Synopsis

```
#include <stats.h>
void autocorrf(a,n,m,c)
const float a[];      /* Input vector a          */
int n;                /* Number of input samples */
int m;                /* Lag count              */
float c[];            /* Output vector c        */
void autocorr_fr16 (a,n,m,c)
const fract16 a[];    /* Input vector a          */
int n;                /* Number of input samples */
int m;                /* Lag count              */
fract16 c[];          /* Output vector c        */
```

Description

This function computes the autocorrelation of the input elements contained within input vector *a*, and stores the result to output vector *c*. The `autocorr` function is used in digital signal processing applications such as speech analysis.

Algorithm

$$c_k = \frac{1}{n} * \left(\sum_{j=0}^{n-k-1} a_j * a_{j+k} \right)$$

where $k = \{0,1,...,m-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for `autocorrf ()`

-1.0 to $+1.0$ for `autocorr_fr16 ()`

cabs

complex absolute value

Synopsis

```
#include <complex.h>
float cabsf (complex_float a);
double cabs (complex_double a);
fract16 cabs_fr16 (fract16 a);
```

Description

This function computes the complex absolute value of a complex input and returns the result.

Algorithm

$$c = \sqrt{\text{Re}^2(a) + \text{Im}^2(a)}$$

Domain

$\text{Re}^2(a) + \text{Im}^2(a) \leq 3.4 \times 10^{38}$ for `cabsf()`, `cabs()`

$\text{Re}^2(a) + \text{Im}^2(a) \leq 1.0$ for `cabs_fr16`

Note

The minimum input value for both real and imaginary parts can be less than 1/256 for `cabs_fr16` but the result may have bit error of 2–3 bits.

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_fract16 cadd_fr16 (complex_fract16 a, complex_fract16 b);
```

Description

This function computes the complex addition of two complex inputs, *a* and *b*, and returns the result.

Algorithm

$$\text{Re}(c) = \text{Re}(a) + \text{Re}(b)$$
$$\text{Im}(c) = \text{Im}(a) + \text{Im}(b)$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for `caddf ()`, `cadd ()`

-1.0 to $+1.0$ for `cadd_fr16 ()`

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_fract16 cdiv_fr16 (complex_fract16 a,
                           complex_fract16 b);
```

Description

This function computes the complex division of complex input a by complex input b, and returns the result.

Algorithm

$$\text{Re}(c) = \frac{\text{Re}(a) * \text{Re}(b) + \text{Im}(a) * \text{Im}(b)}{\text{Re}^2(b) + \text{Im}^2(b)}$$

$$\text{Im}(c) = \frac{\text{Re}(b) * \text{Im}(a) - \text{Im}(b) * \text{Re}(a)}{\text{Re}^2(b) + \text{Im}^2(b)}$$

Domain

-3.4 x 10³⁸ to +3.4 x 10³⁸ for cdivf (), cdiv ()

-1.0 to 1.0 for cdiv_fr16 ()

cexp

complex exponential

Synopsis

```
#include <complex.h>
complex_float cexpf (float a);
complex_double cexp (double a);
```

Description

This function computes the complex exponential of real input *a* and returns the result.

Algorithm

$$\text{Re}(c) = \cos(a)$$
$$\text{Im}(c) = \sin(a)$$

Domain

a = [-9099 ... 9099] for `cexpf ()`, `cexp ()`

cfft

n point radix-2 complex input FFT

Synopsis

```
#include <filter.h>
void cfft_fr16(in[], t[], out[], w[], wst, n, block_exponent,
scale_method)
const complex_fract16 in[]; /* input sequence */
complex_fract16 t[]; /* temporary working buffer */
complex_fract16 out[]; /* output sequence */
const complex_fract16 w[]; /* twiddle sequence */
int wst; /* twiddle factor stride */
int n; /* number of FFT points */
int block_exponent; /* block exponent of output
data */
int scale_method; /* scaling method desired
0-none, 1-static, 2-dynamic */
```

Description

This function transforms the time domain complex input signal sequence to the frequency domain by using the radix-2 Fast Fourier Transform (FFT).

The size of the input array *in*, the output array *out*, and the temporary working buffer *t* is *n*, where *n* represents the number of points in the FFT. By allocating these arrays in different memory banks, any potential data bank collisions will be 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.

DSP Run-Time Library Reference

The twiddle table is passed in the argument `w`, which must contain at least $n/2$ twiddle factors. The function `twidfftrad2_fr16` may be used to initialize the array. If the twiddle table contains more factors than needed for a particular call on `cfft_fr16`, then the stride factor has to be set appropriately; otherwise it should be 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 scales the output by $1/n$.

Algorithm

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk}$$

When the sequence length, n , equals power of four, the `cffttrad4` algorithm is also available.

Domain

Input sequence length n must be a power of two and at least 16.

cffttrad4

n point radix-4 complex input FFT

Synopsis

```

#include <filter.h>
void cffttrad4_fr16 (in[], t[], out[], w[], wst, n,
block_exponent, scale_method)
const complex_fract16 in[]; /* input sequence */
complex_fract16 t[]; /* temporary working buffer */
complex_fract16 out[]; /* output sequence */
const complex_fract16 w[] /* twiddle sequence */
int wst; /* twiddle factor stride */
int n; /* number of FFT points */
int block_exponent; /* block exponent of output
data */
int scale_method; /* scaling method desired
0-none, 1-static, 2-dynamic */

```

Description

This function transforms the time domain complex input signal sequence to the frequency domain by using the radix-4 Fast Fourier Transform. The `cffttrad4_fr16` function will decimate in frequency by the radix-4 FFT algorithm.

The size of the input array `in`, the output array `out`, and the temporary working buffer `t` is `n`, where `n` 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.

DSP Run-Time Library Reference

The twiddle table is passed in the argument `w`, which must contain at least $\frac{3}{4}n$ twiddle coefficients. The function `twidffttrad4_fr16` may be used to initialize the array. If the twiddle table contains more coefficients than needed for a particular call on `cffttrad4_fr16`, then the stride factor has to be set appropriately; otherwise it should be one.

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 `n`.

Algorithm

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk}$$

When the sequence length, `n`, is not a power of four, the radix2 method must be used. See [“cfft” on page 3-35](#).

Domain

Input sequence length `n` must be a power of four and at least 16.

cfft2d

$n \times n$ point 2-d complex input FFT

Synopsis

```
#include <filter.h>

void cfft2d_fr16(*in, *t, *out, w[], wst, n, block_exponent,
scale_method)
const complex_fract16 *in; /* pointer to input matrix a[n][n] */
complex_fract16* t        /* pointer to working buffer t[n][n] */
complex_fract16 *out;     /* pointer to output matrix\ c[n][n] */
const complex_fract16 w[]; /* twiddle sequence */
    int wst;               /* twiddle factor stride */
    int n;                 /* number of FFT points */
    int block_exponent;    /* block exponent of output data */
    int scale_method;      /* caling method desired:
                                0-none, 1-static, 2-dynamic */
```

Description

This function computes the two-dimensional Fast Fourier Transform of the complex input matrix $a[n][n]$ and stores the result to the complex output matrix $c[n][n]$.

The size of the input array *in*, the output array *out*, and the temporary working buffer *t* is *n*, where *n* 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.

DSP Run-Time Library Reference

The twiddle table is passed in the argument `w`, which must contain at least `n` twiddle factors. The function `twidfft2d_fr16` may be used to initialize the array. If the twiddle table contains more factors than needed for a particular call on `cfft2d_fr16`, then the stride factor has to be set appropriately; otherwise it should be 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 scales the output by $n*n$.

Algorithm

$$c(i, j) = \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} a(k, l) * e^{-2\pi j(i*k + j*l)/n}$$

where $i=\{0,1,\dots,n-1\}$, $j=\{0,1,2,\dots,n-1\}$

Domain

Input sequence length `n` must be a power of two and at least 16.

cfir

complex finite impulse response filter

Synopsis

```
#include <filter.h>

void cfir_fr16(x,y,n,s)
const complex_fract16 x[]; /* Input sample vector x */
complex_fract16 y[];      /* Output sample vector y */
int n;                    /* Number of input samples */
cfir_state_fr16 *s;       /* Pointer to filter state
                           structure */
```

To maintain the state of the filter, the function uses:

```
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;
```

Description

The `cfir_fr16` function implements a complex finite impulse response (CFIR) filter. It generates the filtered response of the complex input data `x` and stores the result in the complex output vector `y`.

The function maintains the filter state in the structured variable `s`, 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 and is defined as:

DSP Run-Time Library Reference

```
#define fir_init(state, coeffs, delay, ncoeffs) \  
    (state).h = (coeffs); \  
    (state).d = (delay); \  
    (state).p = (delay); \  
    (state).k = (ncoeffs)
```

The characteristics of the filter (passband, stopband, etc.) are dependent upon the number of complex filter coefficients and their values. A pointer to the coefficients should be stored in `s->h`, and `s->k` should be set to the number of coefficients.

Each filter should have its own delay line which is a vector of type `complex_fract16` and 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 `s->d` should be set to the start of the delay line, and the function uses `s->p` to keep track of its current position within the vector.

Algorithm

$$y(k) = \sum_{i=0}^{p-1} h(i) * x(k-i) \text{ for } k = 0, 1, \dots, n$$

Domain

−1.0 to +1.0

clip

clip

Synopsis

```
#include <math.h>
int clip (int parm1, int parm2)
float fclipf (float parm1, float parm2)
double fclip (double parm1, double parm2)
fract16 clip_fr16 (fract16 parm1, fract16 parm2)
```

Description

This function clips a value if it is too large.

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_fract16 cmlt_fr16 (complex_fract16 a, complex_fract16 b)
```

Description

This function computes the complex multiply of two complex inputs, *a* and *b*, and returns the result.

Algorithm

$$\begin{aligned} \text{Re}(c) &= \text{Re}(a) * \text{Re}(b) - \text{Im}(a) * \text{Im}(b) \\ \text{Im}(c) &= \text{Re}(a) * \text{Im}(b) + \text{Im}(a) * \text{Re}(b) \end{aligned}$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for `cmltf ()`, `cmlt ()`

-1.0 to 1.0 for `cmlt_fr16 ()`

conj

complex conjugate

Synopsis

```
#include <complex.h>
complex_float conjf (complex_float a)
complex_double conj (complex_double a)
complex_fract16 conj_fr16 (complex_fract16 a)
```

Description

This function conjugates the complex input *a* and returns the result.

Algorithm

$$\begin{aligned}\text{Re}(c) &= \text{Re}(a) \\ \text{Im}(c) &= -\text{Im}(a)\end{aligned}$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for `conjf ()`, `conj ()`

-1.0 to 1.0 for `conj_fr16 ()`

convolve

convolution

Synopsis

```
#include <filter.h>
void convolve_fr16(cin1, clen1, cin2, clen2, cout)
const fract16 cin1[];    /* pointer to input sequence 1 */
int clen1;               /* length of the input sequence 1 */
const fract16 cin2[];    /* pointer to input sequence 2 */
int clen2;               /* length of the input sequence 2 */
fract16 cout[];          /* pointer to output sequence */
```

Description

This function convolves two sequences pointed to by `cin1` and `cin2`. If `cin1` points to the sequence whose length is `clen1` and `cin2` points to the sequence whose length is `clen2`, the resulting sequence pointed to by `cout` has length `clen1 + clen2 - 1`.

Algorithm

Convolution between two sequences `cin1` and `cin2` is described as:

$$cout[n] = \sum_{k=0}^{clen2-1} cin1[n+k-(clen2-1)] \bullet cin2[(clen2-1)-k]$$

for $n = 0$ to $clen1 + clen2 - 1$. Values for `cin1[j]` are considered to be zero for $j < 0$ or $j > clen1 - 1$.

Example

Here is an example of a convolution where `cin1` is of length 4 and `cin2` is of length 3. If we represent `cin1` as “A” and `cin2` 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]}
```

Domain

−1.0 to +1.0

conv2d

2-d convolution

Synopsis

```
#include <filter.h>
void conv2d_fr16(min1, mrow1, mcol1, min2, mrow2, mcol2, mout )
const fract16 *min1; /* pointer to input matrix 1 */
int mrow1;           /* number of rows in matrix 1 */
int mcol1;           /* number of columns in matrix 1 */
const fract16 *min2; /* pointer to input matrix 2 */
int mrow2;           /* number of rows in matrix 2 */
int mcol2;           /* number of columns in matrix 2 */
fract16 *mout;       /* pointer to output matrix */
```

Description

This function computes two-dimensional convolution of input matrix `min1` of size `mrow1 x mcol1` and `min2` of size `mrow2 x mcol2` and stores the result in matrix `mout` of dimension $(mrow1 + mrow2 - 1) \times (mcol1 + mcol2 - 1)$.



A temporary work area is allocated from the run-time stack the function uses 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

Algorithm

Two dimensional input matrix `min1` is convolved with input matrix `min2`, placing the result in a matrix pointed to by the `mout`.

$$mout[c, r] = \sum_{i=0}^{mcol2-1} \sum_{j=0}^{mrow2-1} min1[c+i, r+j] \bullet min2[(mcol2-1)-i, (mrow2-1)-j]$$

for $c = 0$ to $mcol1+mcol2-1$ and $r = 0$ to $mrow2-1$

Domain

-1.0 to $+1.0$

conv2d3x3

2-d convolution with 3 x 3 matrix

Synopsis

```
#include <filter.h>
void conv2d3x3_fr16(min1, mrow1, mcol1, min2, mout )
const fract16 *min1;      /* pointer to input matrix 1      */
int mrow1;                 /* number of rows in matrix 1    */
int mcol1;                 /* number of columns in matrix 1 */
const fract16 *min2;      /* pointer to input matrix 2      */
fract16 *mout;             /* pointer to output matrix       */
```

Description

This function computes two-dimensional convolution of matrix `min1` (size `mrow1` x `mcol1`) with matrix `min2` (size 3 x 3).

Algorithm

Two dimensional input matrix `min1` is convolved with input matrix `min2`, placing the result in a matrix pointed to by `mout`.

$$mout [c, r] = \sum_{i=0}^2 \sum_{j=0}^2 min1 [c + i, r + j] \bullet min2 [2 - i, 2 - j]$$

for `c = 0` to `mcol1+2` and `r = 0` to `mrow1+2`

Domain

-1.0 to +1.0

copysign

copysign

Synopsis

```
#include <math.h>
float copysignf (float parm1, float parm2)
double copysign (double parm1, double parm2)
fract16 copysign_fr16 (fract16 parm1, fract16 parm2)
```

Description

Copies 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)
```

Description

This function calculates the cotangent of its argument *a*, which is measured in radians. If *a* is outside of the domain, the function returns 0.

Algorithm

```
c = cot(a)
```

Domain

```
x = [-9099 ... 9099]
```

countones

count one bits in word

Synopsis

```
#include <math.h>
int countones(int word)
int lcountones(long word)
```

Description

This function counts the number of one bits in a word.

Algorithm

$$\text{return} = \sum_{j=0}^{j=31} \text{bit}[j] \text{ of word}$$

crosscoh

cross-coherence

Synopsis

```
#include <stats.h>
void crosscohf(a,b,n,m,c)
const float a[]; /* Input vector a */
const float b[]; /* Input vector b */
int n; /* Number of input samples */
int m; /* Lag count */
float c[]; /* Output vector c */
void crosscoh _fr16(a,n,m,c)
const fract16 a[]; /* Input vector a */
const fract16 b[]; /* Input vector b */
int n; /* Number of input samples */
int m; /* Lag count */
fract16 c[]; /* Output vector c */
```

Description

This function computes the cross-coherence of the input elements contained within input vector *a* and input vector *b*. It stores the result to output vector *c*.

Algorithm

$$c_k = \frac{1}{n} * \left(\sum_{j=0}^{n-k-1} (a_j - \bar{a}) * (b_{j+k} - \bar{b}) \right)$$

where $k = \{0, 1, \dots, m-1\}$, \bar{a} is the mean value of input vector *a*, and \bar{b} is the mean value of input vector *b*.

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for crosscohf ()

−1.0 to +1.0 for `crosscoh_fr16 ()`

crosscorr

cross-correlation

Synopsis

```
#include <stats.h>
void crosscorrfrf(a,b,n,m,c)
const float a[];          /* Input vector a          */
const float b[];          /* Input vector b          */
int n;                    /* Number of input samples */
int m;                    /* Lag count               */
float c[];                /* Pointer to output vector c */
void crosscorr_fr16(a,n,m,c)
const fract16 a[];        /* Input vector a          */
const fract16 b[];        /* Input vector b          */
int n;                    /* Number of input samples */
int m;                    /* Lag count               */
fract16 c[];              /* Pointer to output vector c */
```

Description

This function computes the cross-correlation of the input elements contained within input vector a and input vector b. It stores the result to output vector c.

Algorithm

$$c_k = \frac{1}{n} * \left(\sum_{j=0}^{n-k-1} a_j * b_{j+k} \right)$$

where k = {0,1,...,m-1}

Domain

-3.4 x 10³⁸ to +3.4 x 10³⁸ for crosscorrfrf ()

-1.0 to +1.0 for crosscorr_fr16 ()

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_fract16 csub_fr16 (complex_fract16 a,
                           complex_fract16 b);
```

Description

This function computes the complex subtraction of two complex inputs, *a* and *b*, and returns the result.

Algorithm
$$\text{Re}(c) = \text{Re}(a) - \text{Re}(b)$$
$$\text{Im}(c) = \text{Im}(a) - \text{Im}(b)$$
Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for `csubf ()`, `csub ()`

-1.0 to 1.0 for `csub_fr16 ()`

fir

finite impulse response filter

Synopsis

```
#include <filter.h>

void fir_fr16(x,y,n,s)
const fract16 x[];      /* Input sample vector x          */
fract16 y[];            /* Output sample vector y        */
int n;                  /* Number of input samples       */
fir_state_fr16 *s;      /* Pointer to filter state structure */
```

To maintain the state of the filter, the FIR (Finite Impulse Response) function uses:

```
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;
```

Description

The `fir_fr16` function implements a FIR filter. The function generates the filtered response of the input data `x` and stores the result in the output vector `y`.

The function maintains the filter state in the structured variable `s` which must be declared and initialized before calling the function. The macro `fir_init`, in the `filter.h` header file, is available 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 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 `s->h`, and `s->k` should be set to the number of coefficients.

Each filter should have its own delay line which is a vector of type `fract16` and 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 `s->d` should be set to the start of the delay line, and the function uses `s->p` to keep track of its current position within the vector.

The structure member `s->l` is not used by `fir_fr16`. This field is normally set to an interpolation/decimation index before calling either the `fir_interp_fr16` or `fir_decima_fr16` functions.

Algorithm

$$y(i) = \sum_{j=0}^{k-1} h(j) * x(j-i) \text{ for } i=0,1,..n-1$$

Domain

−1.0 to +1.0

fir_decima

FIR decimation filter

Synopsis

```
#include <filter.h>

void fir_decima_fr16(x,y,n,s)
const fract16 x[];      /* Input sample vector x          */
fract16 y[];            /* Output sample vector y          */
int n;                  /* Number of samples to generate   */
fir_state_fr16 *s;      /* Pointer to filter state structure */
```

To maintain the state of the filter, the function uses:

```
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;
```

Description

The `fir_decima_fr16` function performs a FIR-based decimation filter. It generates the filtered decimated response of the input data `x` and stores the result in the output vector `y`.

The function maintains the filter state in the structured variable `s`, which must be declared and initialized before calling the function. The macro `fir_init`, in the `filter.h` header file, is available 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 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 `s->h`, and `s->k` should be set to the number of coefficients. The decimation index is supplied to the function in `s->l`.

Each filter should have its own delay line which is a vector of type `fract16` and 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 `s->d` should be set to the start of the delay line, and the function uses `s->p` to keep track of its current position within the vector.

Algorithm

$$y(k) = \sum_{i=0}^{p-1} x(k * l - i) * h(i)$$

Domain

−1.0 to + 1.0

fir_interp

FIR interpolation filter

Synopsis

```
#include <filter.h>

void fir_interp_fr16(x,y,n,s)
const fract16 x[];      /* Input sample vector x          */
fract16 y[];            /* Output sample vector y          */
int n;                  /* Number of input samples         */
fir_state_fr16 *s;      /* Pointer to filter state structure */
```

To maintain the state of the filter, the function uses:

```
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;
```

Description

The `fir_interp_fr16` function performs a FIR-based interpolation filter. It generates the interpolated filtered response of the input data `x` and stores the result in the output vector `y`.

The function maintains the filter state in the structured variable `s`, which must be declared and initialized before calling the function. The macro `fir_init`, in the `filter.h` header file, is available 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 characteristics of the filter are dependent upon the number of polyphase filter coefficients and their values, and on the interpolation index supplied by the calling program. A pointer to the coefficients should be stored in `s->h`, and `s->k` should be set to the number of coefficients. The interpolation index is supplied to the function in `s->l`.

Each filter should have its own delay line which is a vector of type `fract16` and 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 `s->d` should be set to the start of the delay line, and the function uses `s->p` to keep track of its current position within the vector.

Algorithm

$$y_m(k) = \sum_{i=0}^{p/l-1} x(k-i) * h(i * l + m)$$

where $m = \{0, 1, 2, \dots, l\}$

Domain

−1.0 to +1.0

gen_bartlett

generate Bartlett window

Synopsis

```
#include <window.h>
void gen_bartlett_fr16(w,a,N)
fract16 w[]; /* Window vector */
int a; /* Address stride in samples for window vector */
int N; /* Length of window vector */
```

Description

This function generates a vector containing the Bartlett window. The length is specified by parameter N. This window is similar to the Triangle window but has two 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 a N+2 Bartlett window equals a 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

$$w[n] = 1 - \left| \frac{n - \frac{N-1}{2}}{\frac{N-1}{2}} \right|$$

where $n = \{0, 1, 2, \dots, N-1\}$

Domain

$a > 0$; $N > 0$

gen_blackman

generate Blackman window

Synopsis

```
#include <window.h>
void gen_blackman_fr16(w,a,N)
fract16 w[]; /* Window vector */
int a; /* Address stride in samples for window vector */
int N; /* Length of window vector */
```

Description

This function generates a vector containing the Blackman window. The length is specified by parameter N.

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 $n = \{0, 1, 2, \dots, N-1\}$

Domain

$a > 0$; $N > 0$

gen_gaussian

generate Gaussian window

Synopsis

```
#include <window.h>
void gen_gaussian_fr16(w,alpha,a,N)
fract16 w[]; /* Window vector */
float alpha; /* Gaussian alpha parameter */
int a; /* Address stride in samples for window vector */
int N; /* Length of window vector */
```

Description

This function generates a vector containing the Gaussian window. The length is specified by parameter N.

Algorithm

$$w(n) = \exp \left[-\frac{1}{2} \left(\alpha \frac{n - N/2 - 1/2}{N/2} \right)^2 \right]$$

where $n = \{0, 1, 2, \dots, N-1\}$ and α is an input parameter.

Domain

$a > 0$; $N > 0$; $\alpha > 0.0$

gen_hamming

generate Hamming window

Synopsis

```

#include <window.h>
void gen_hamming_fr16(w,a,N)
fract16 w[];    /* Window vector */
int a;          /* Address stride in samples for window vector */
int N;          /* Length of window vector */

```

Description

This function generates a vector containing the Hamming window. The length is specified by parameter N.

Algorithm

$$w[n] = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N-1}\right)$$

where $n = \{0, 1, 2, \dots, N-1\}$

Domain

$a > 0$; $N > 0$

gen_hanning

generate Hanning window

Synopsis

```
#include <window.h>
void gen_hanning_fr16(w,a,N)
fract16 w[]; /* Window vector */
int a; /* Address stride in samples for window vector */
int N; /* Length of window vector */
```

Description

This function generates a vector containing the Hanning window. The length is specified by parameter N. This window is also known as the Cosine window.

Algorithm

$$w[n] = 0.5 - 0.5 \cos\left(\frac{2\pi n}{N+1}\right)$$

where $n = \{0, 1, 2, \dots, N+1\}$

Domain

$a > 0$; $N > 0$

gen_harris

generate Harris window

Synopsis

```
#include <window.h>
void gen_harris_fr16(w,a,N)
fract16 w[];    /* Window vector */
int a;          /* Address stride in samples for window vector */
int N;          /* Length of window vector */
```

Description

This function generates a vector containing the Harris window. The length is specified by parameter N. This window is also known as the Blackman-Harris window.

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 $n = \{0, 1, 2, \dots, N-1\}$

Domain

$a > 0$; $N > 0$

gen_kaiser

generate Kaiser window

Synopsis

```
#include <window.h>
void gen_kaiser_fr16(w,beta,a,N)
fract16 w[]; /* Window vector */
float beta; /* Kaiser beta parameter */
int a; /* Address stride in samples for window vector */
int N; /* Length of window vector */
```

Description

This function generates a vector containing the Kaiser window. The length is specified by parameter N. The β value is specified by parameter b.

Algorithm

$$w[n] = \frac{I_0 \left[\beta \left(1 - \left[\frac{n - \alpha}{\alpha} \right]^2 \right)^{1/2} \right]}{I_0(\beta)}$$

where $n = \{0, 1, 2, \dots, N-1\}$, $\alpha = (N - 1) / 2$, and $I_0(\beta)$ represents the zeroth-order modified Bessel function of the first kind.

Domain

$\alpha > 0$; $N > 0$; $\beta > 0.0$

gen_rectangular

generate rectangular window

Synopsis

```
#include <window.h>
void gen_rectangular_fr16(w,a,N)
fract16 w[]; /* Window vector */
int a; /* Address stride in samples for window vector */
int N; /* Length of window vector */
```

Description

This function generates a vector containing the Rectangular window. The length is specified by parameter N.

Algorithm

$w[n] = 1$ where $n = \{0, 1, 2, \dots, N-1\}$

Domain

$a > 0$; $N > 0$

gen_triangle

generate triangle window

Synopsis

```
#include <window.h>
void gen_triangle_fr16(w,a,N)
fract16 w[]; /* Window vector */
int a; /* Address stride in samples for window vector */
int N; /* Length of window vector */
```

Description

This function generates a vector containing the Triangle window. The length is specified by parameter N. Refer to the Bartlett window regarding the relationship between it and the Triangle window.

Algorithm

For even n:

$$w[n] = \begin{cases} \frac{2n+1}{N} & n < N/2 \\ \frac{2N-2n-1}{N} & n > N/2 \end{cases}$$

where $n = \{0, 1, 2, \dots, N-1\}$

For odd n:

$$w[n] = \begin{cases} \frac{2n+2}{N+1} & n < N/2 \\ \frac{2N-2n}{N+1} & n > N/2 \end{cases}$$

where $n = \{0, 1, 2, \dots, N-1\}$

Domain

$a > 0; N > 0$

gen_vonhann

generate Von Hann window

Synopsis

```
#include <window.h>
void gen_vonhann_fr16(w,a,N)
fract16 w[]; /* Window vector */
int a;      /* Address stride in samples for window vector */
int N;      /* Length of window vector */
```

Description

This function is identical to the Hanning window.

Domain

$a > 0$; $N > 0$

histogram

histogram

Synopsis

```
#include <stats.h>
void histogramf(a,c,max,min,n,m)
const float a[];      /* Pointer to input vector a */
int c[];              /* Pointer to output vector c */
float max;            /* Maximum value of the bin */
float min;            /* Minimum value of the bin */
int n;                /* Number of input samples */
int m;                /* Number of bins */
void histogram_fr16(a,n,m,c)
const fract16 a[];    /* Pointer to input vector a */
int c[];              /* Pointer to output vector c */
fract16 max;          /* Maximum value of the bin */
fract16 min;          /* Minimum value of the bin */
int n;                /* Number of input samples */
int m;                /* Number of bins */
```

Description

This function computes the histogram of the input elements contained within input vector *a*, and stores the result to output vector *c*. It bins the element of input vector *x* into *m* equally spaced containers, and returns the number of elements in each container.

Domain

−3.4 x 10³⁸ to +3.4 x 10³⁸ for histogramf ()

−1.0 to +1.0 for histogram_fr16 ()

ifft

n point radix-2 inverse FFT

Synopsis

```
#include <filter.h>
void ifft_fr16(in[], t[], out[], w[], wst, n, block_exponent,
scale_method)
const complex_fract16 in[]; /* input sequence */
complex_fract16 t[]; /* temporary working buffer */
complex_fract16 out[]; /* output sequence */
const complex_fract16 w[]; /* twiddle sequence */
int wst; /* twiddle factor stride */
int n; /* number of FFT points */
int block_exponent; /* block exponent of output data */
int scale_method; /* scaling method desired
0-none, 1-static, 2-dynamic */
```

Description

This function transforms 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 `in`, the output array `out`, and the temporary working buffer `t` is `n`, where `n` represents the number of points in the FFT. To avoid potential data bank collisions the input and temporary buffers should be allocated in different memory banks; this will result 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.

The twiddle table is passed in the argument `w`, which must contain at least $n/2$ twiddle coefficients. The function `twidffttrad2_fr16` may be used to initialize the array. If the twiddle table contains more coefficients than needed for a particular call on `ifft_fr16`, then the stride factor has to be set appropriately; otherwise it should be 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 scales the output by $1/n$.

Algorithm

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-nk}$$

The implementation uses core FFT functions. To get the inverse effect, it first swaps the real and imaginary parts of the input, performs the direct radix-2 transformation, and finally swaps the real and imaginary parts of the output.

Domain

Input sequence length `n` must be a power of two and at least 16.

iffttrad4

n point radix-4 inverse input FFT

Synopsis

```
#include <filter.h>
void iffttrad4_fr16 (in[], t[], out[], w[], wst, n,
block_exponent, scale_method)
const complex_fract16 in[]; /* input sequence */
complex_fract16 t[]; /* temporary working buffer */
complex_fract16 out[]; /* output sequence */
const complex_fract16 w[]; /* twiddle sequence */
int wst; /* twiddle factor stride */
int n; /* number of FFT points */
int block_exponent; /* block exponent of output data */
int scale_method; /* scaling method desired:
0-none, 1-static, 2-dynamic */
```

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 `in`, the output array `out`, and the temporary working buffer `t` is `n`, where `n` 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 `w`, which must contain at least $\frac{3}{4}n$ 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 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 `n`.

Algorithm

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-nk}$$

The implementation uses core FFT functions. To get the inverse effect, the function first swaps the real and imaginary parts of the input, performs the direct radix-4 transformation, and finally swaps the real and imaginary parts of the output.

Domain

Input sequence length `n` must be a power of four and at least 16.

ifft2d

n x n point 2-d inverse input FFT

Synopsis

```
#include <filter.h>
void ifft2d_fr16(*in, *t, *out, w[], wst, n, block_exponent,
scale_method)
const complex_float *in; /* pointer to input matrix a[n][n] */
complex_fract16 *t;      /* pointer to working buffer t[n][n] */
complex_fract16 *out;    /* pointer to output matrix c[n][n] */
const complex_fract16 w[]; /* twiddle sequence */
int wst;                 /* twiddle factor stride */
int n;                   /* number of FFT points */
int block_exponent;      /* block exponent of output data */
int scale_method;        /* scaling method desired:
                           0-none, 1-static, 2-dynamic */
```

Description

This function computes a two-dimensional Inverse Fast Fourier Transform of the complex input matrix $a[n][n]$ and stores the result to the complex output matrix $c[n][n]$.

The size of the input array *in*, the output array *out*, and the temporary working buffer *t* is *n*, where *n* 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 `w`, which must contain at least `n` twiddle factors. The function `twidfft2d_fr16` may be used to initialize the array. If the twiddle table contains more factors than needed for a particular call on `ifft2d_fr16`, then the stride factor has to be set appropriately; otherwise it should be one.

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 `n*n`.

Algorithm

$$c(i, j) = \frac{1}{n^2} \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} a(k, l) * e^{2\pi j(i*k + j*l)/n}$$

where $i=\{0,1,...,n-1\}$, $j=\{0,1,2,...,n-1\}$

Domain

Input sequence length `n` must be a power of two and at least 16.

iir

infinite impulse response filter

Synopsis

```
#include <filter.h>

void iir_fr16(x,y,n,s)
const fract16 x[];      /* Input sample vector x          */
fract16 y[];             /* Output sample vector y        */
int n;                   /* Number of input samples       */
iir_state_fr16 *s;       /* Pointer to filter state structure */
```

to maintain the state of the filter, the IIR function uses:

```
typedef struct
{
    fract16 *c;           /* coefficients                */
    fract16 *d;           /* start of delay line         */
    int k;                 /* number of bi-quad stages    */
} iir_state_fr16;
```

Description

The `iir_fr16` function implements a bi-quad, canonical form, infinite impulse response (IIR) filter. It generates the filtered response of the input data `x` and stores the result in the output vector `y`.

The function maintains the filter state in the structured variable `s`, which must be declared and initialized before calling the function. The macro `iir_init`, 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. A pointer to the coefficients should be stored in `s->c`, and `s->k` should be set to the number of stages.

Each filter should have its own delay line which is a vector of type `fract16` and 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 `s->d` should be set to the start of the delay line.

Algorithm

$$H(z) = \frac{B_0 + B_1 z^{-1} + B_2 z^{-2}}{1 - A_1 z^{-1} - A_2 z^{-2}}$$

where

$$\begin{aligned} D_m &= A_2 * D_{m-2} + A_1 * D_{m-1} + x_m \\ Y_m &= B_2 * D_{m-2} + B_1 * D_{m-1} + B_0 * D_m \end{aligned}$$

where $m = \{0, 1, 2, \dots, n-1\}$

Domain

-1.0 to +1.0

DSP Run-Time Library Reference

max

maximum

Synopsis

```
#include <math.h>
int max (int parm1, int parm2)
float fmaxf (float parm1, float parm2)
double fmax (double parm1, double parm2)
fract16 max_fr16 (fract16 parm1, fract16 parm2)
```

Description

This function returns the larger of its two arguments.

Algorithm

```
if ( parm1 > parm2)
    return( parm1)
else
    return( parm2)
```

Domain

Full range for type of parameters.

mean

mean

Synopsis

```
#include <stats.h>
fractl6 mean_fr16(a,n)
const fractl6 a[];      /* Input vector a          */
int n;                  /* Number of input samples */
float meanf(a,n)
const float a[];        /* Input vector a          */
int n;                  /* Number of input samples */
```

Description

This function computes the mean of the input elements contained within input vector *a* and returns the result.

Algorithm

$$c = \frac{1}{n} * \left(\sum_{i=0}^{n-1} a_i \right)$$

Domain

−3.4 x 10³⁸ to +3.4 x 10³⁸ for meanf ()

−1.0 to +1.0 for mean_fr16 ()

min

minimum

Synopsis

```
#include <math.h>
int min (int parm1, int parm2)
float fminf (float parm1, float parm2)
double fmin (double parm1, double parm2)
fract16 min_fr16 (fract16 parm1, fract16 parm2)
```

Description

This function returns the smaller of its 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(in, out, n)
const short in[];    /* Input array */
short out[];         /* Output array */
int n;               /* Number of elements to be compressed */
```

Description

This function performs μ-law compression (ITU rec. G.711) on the elements of the input vector pointed to by `in` and outputs compressed data in the vector pointed to by `out`.

Algorithm

$C(k)$ = mu_law compression of $A(k)$ for $k=0$ to $n-1$

Domain

Content of input array: -8192 to 8191

mu_expand

μ-law expansion

Synopsis

```
#include <filter.h>
void mu_expand(in, out, n)
const short in[]; /* Input array */
short out[]; /* Output array */
int n; /* Number of elements to be expanded */
```

Description

This function performs μ-law expansion (ITU rec. G.711) on the elements of the input vector pointed to by *in* and outputs expanded data in the vector pointed to by *out*.

Algorithm

$C(k) = \text{mu_law expansion of } A(k) \text{ for } k=0 \text{ to } n-1$

Domain

Content of input array: 0 to 255

norm

normalization

Synopsis

```
#include <complex.h>
complex_float normf (complex_float a)
complex_double norm (complex_double a)
```

Description

This function normalizes the complex input *a* and returns the result.

Algorithm

$$\text{Re}(c) = \frac{\text{Re}(a)}{\sqrt{\text{Re}^2(a) + \text{Im}^2(a)}}$$
$$\text{Im}(c) = \frac{\text{Im}(a)}{\sqrt{\text{Re}^2(a) + \text{Im}^2(a)}}$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

polar

construct from polar coordinates

Synopsis

```
#include <complex.h>
complex_float polarf (float mag, float phase)
complex_double polar (double mag, double phase)
complex_fract16 polar_fr16 (fract16 mag, fract16 phase)
```

Description

This function transforms the polar coordinate to normal coordinate.

Algorithm

$$\text{Re}(c) = r \cdot \cos(\theta)$$
$$\text{Im}(c) = r \cdot \sin(\theta)$$

where θ is the phase, and r is the magnitude

Domain

phase = $[-9099 \dots 9099]$ for `polarf()`, `polar()`
mag = -3.4×10^{38} to $+3.4 \times 10^{38}$ for `polarf()`, `polar()`
phase = -1.0 to $+1.0$ for `polar_fr16()`
mag = -1.0 to $+1.0$ for `polar_fr16()`

rfft

n point radix-2 real input FFT

Synopsis

```
#include <filter.h>
void rfft_fr16(in[], t[], out[], w[], wst, n, block_exponent,
scale_method)
const fract16 in[];           /* input/output sequence      */
complex_fract16 t[];          /* temporary working buffer   */
complex_fract16 out[];        /* working buffer             */
const complex_fract16 w[];    /* twiddle sequence           */
int wst;                      /* twiddle factor stride      */
int n;                        /* number of FFT points       */
int block_exponent;          /* block exponent of output data */
int scale_method;            /* scaling method desired:
                                0-none, 1-static, 2-dynamic */
```

Description

This function transforms the time domain real input signal sequence to the frequency domain by using the radix-2 FFT. The 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 `in`, the output array `out`, and the temporary working buffer `t` is `n`, where `n` 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.

DSP Run-Time Library Reference

The twiddle table is passed in the argument `w`, which must contain at least $n/2$ twiddle factors. The function `twidffttrad2_fr16` may be used to initialize the array. If the twiddle table contains more factors than needed for a particular call on `rfft_fr16`, then the stride factor has to be set appropriately; otherwise it should be 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 $1/n$.

Algorithm

See “[cfft](#)” on page 3-35 for more information. Output sequence will have $n/2$ complex members due to a fact that FFT of real input data produces symmetric output around half-sampling frequency point.

Domain

Input sequence length n must be a power of two and at least 16.

rfftrad4

n point radix-4 real input FFT

Synopsis

```

#include <filter.h>
void rfftrad4_fr16(in[], t[], out[], w[], wst, n, block_exponent,
scale_method)
const fract16 in[];           /* input/output sequence      */
complex_fract16 t[];          /* temporary working buffer   */
complex_fract16 out[];        /* working buffer             */
const complex_fract16 w[];    /* twiddle sequence           */
int wst;                      /* twiddle factor stride      */
int n;                        /* number of FFT points       */
int block_exponent;          /* block exponent of output data */
int scale_method;            /* scaling method desired:
                                0-none, 1-static, 2-dynamic */

```

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 `in`, the output array `out`, and the temporary working buffer `t` is `n`, where `n` 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 will result 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.

DSP Run-Time Library Reference

The twiddle table is passed in the argument `w`, which must contain at least $\frac{3}{4}n$ 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 one.

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 `n`.

Algorithm

See “[cffttrad4](#)” on page 3-37 for more information. Output sequence will have $n/2$ complex members due to a fact that FFT of real input data produces symmetric output around half-sampling frequency point.

Domain

Input sequence length `n` must be a power of four and at least 16.

rfft2d

$n \times n$ point 2-d real input FFT

Synopsis

```

#include <filter.h>
void rfft2d_fr16(*in, *t, *out, w[], wst, n, block_exponent,
scale_method)
const fract16 *in;           /* pointer to input matrix a[n][n] */
complex_fract16 *t;          /* pointer to working buffer t[n][n] */
complex_fract16 *out;        /* pointer to output matrix [n][n] */
const complex_fract16 w[];   /* twiddle sequence */
int wst;                     /* twiddle factor stride */
int n;                       /* number of FFT points */
int block_exponent;          /* block exponent of output data */
int scale_method;            /* scaling method desired:
                                0-none, 1-static, 2-dynamic */

```

Description

This function computes a two-dimensional Fast Fourier Transform of the real input matrix $a[n][n]$, and stores the result to the complex output matrix $c[n][n]$.

The size of the input array in , the output array out , and the temporary working buffer t is n , where n represents the number of points in the FFT. Improved run-time performance can be achieved by allocating the input and temporary arrays in separate memory banks; this will avoid any memory bank collisions. 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 w , which must contain at least n twiddle coefficients. The function `twidfft2d_fr16` may be used to initialize the array. If the twiddle table contains more coefficients than needed for a particular call on `rfft2d_fr16`, then the stride factor has to be set appropriately; otherwise it should be one.

DSP Run-Time Library Reference

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 scales the output by $n*n$.

Algorithm

$$c(i, j) = \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} a(k, l) * e^{-2\pi j(i*k + j*l)/n}$$

where $i=\{0,1,...,n-1\}$, $j=\{0,1,2,...,n-1\}$

Domain

Input sequence length n must be a power of two and at least 16.

rms

root mean square

Synopsis

```
#include <stats.h>
float rmsf(a,n)
const float a[];      /* Pointer to input vector a */
int n;                /* Number of input samples */
fract16 rms_fr16(a,n)
const fract16 a[];    /* Pointer to input vector a */
int n;                /* Number of input samples */
```

Description

This function computes the root mean square of the input elements contained within input vector *a* and returns the result.

Algorithm

$$c = \sqrt{\frac{\sum_{i=0}^{n-1} a_i^2}{n}}$$

Domain

−3.4 x 10³⁸ to +3.4 x 10³⁸ for rmsf ()

−1.0 to +1.0 for rms_fr16 ()

rsqrt

reciprocal square root

Synopsis

```
#include <math.h>
float rsqrtf (float a)
double rsqrt (double a)
```

Description

This function calculates the reciprocal of the square root of the number *a*.
If *a* is negative, the function returns 0.

Algorithm

$$c = 1 / \sqrt{a}$$

Domain

[0.0 ... 3.4 x 10³⁸]

twidfftrad2

generate FFT twiddle factors for radix-2 FFT

Synopsis

```
#include <filter.h>
void twidfftrad2_fr16 (complex_fract16 w[], int n)
```

Description

This function calculates complex twiddle coefficients for a radix-2 FFT with n points and returns the coefficients in the vector w . The vector w , known as the twiddle table, is normally calculated once and is then passed to an FFT function as a separate argument. The size of the table must be $\frac{1}{2}$ of n , the number of points in the FFT.

FFTs of different sizes can be accommodated with the same twiddle table. Simply allocate the table at the maximum size. Each FFT has an additional parameter, the “stride” of the twiddle table. To use the whole table, specify a stride of 1. If the FFT uses only half the points of the largest element, the stride should be 2 (this takes only every other element).

Algorithm

This function takes FFT length n 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 $k = \{0, 1, 2, \dots, n/2 - 1\}$

DSP Run-Time Library Reference

Domain

The FFT length n must be a power of two and at least 16.

twidfftrad4

generate FFT twiddle factors for radix-4 FFT

Synopsis

```
#include <filter.h>
void twidfftrad4_fr16 (complex_fract16 w[], int n)
void twidfft_fr16(complex_fract16 w[], int n)
```

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 `n`, and the coefficients are returned in the twiddle table `w`.

The size of the twiddle table must be $\frac{3}{4}n$, 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 uses only half the number of points of the largest FFT, the stride should be 2.

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 `twidfft_fr16` routine is provided as an alternative to the `twidfftrad4_fr16` routine and performs the same function.

Algorithm

This function takes FFT length `n` as an input parameter and generates the lookup table of complex twiddle coefficients.

DSP Run-Time Library Reference

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 $k = \{0, 1, 2, \dots, \frac{3}{4}n - 1\}$

Domain

The FFT length n must be a power of two and at least 16.

twidfft2d

generate FFT twiddle factors for 2-D FFT

Synopsis

```
#include <filter.h>
void twidfft2d_fr16 (complex_fract16 w[], int n)
```

Description

The `twidfft2d_fr16` function generates complex twiddle factors for a 2-D FFT. The size of the FFT input sequence is given by the argument `n` and the function writes the twiddle factors to the vector `w`, known as the twiddle table.

The size of the twiddle table must be `n`, the number of points in the FFT. Normally, the table is only calculated once and is then passed to an FFT function as an argument. A twiddle table may be used to generate several FFTs of different sizes by initializing the table for the largest FFT and then using the stride argument of the FFT function to specify the step size through the table. For example, to generate the largest FFT, the stride would be set to 1, and to generate an FFT of half this size the stride would be set to 2.

Algorithm

This function takes FFT length `n` as an input parameter and generates the lookup table of complex twiddle coefficients.

The samples generated are:

$$twid_re(k) = \cos\left(\frac{2\pi}{n}k\right)$$

$$twid_im(k) = \sin\left(\frac{2\pi}{n}k\right)$$

DSP Run-Time Library Reference

where $k = \{0, 1, 2, \dots, n-1\}$

Domain

The FFT length n must be a power of two and at least 16.

var

variance

Synopsis

```
#include <stats.h>
float varf(a,n)
const float a[];          /* Pointer to input vector a */
int n;                    /* Number of input samples */
fract16 var_fr16(a, n)
const fract16 a[];        /* Pointer to input vector a */
int n;                    /* Number of input samples */
```

Description

This function computes the variance of the input elements contained within input vector a and returns the result.

Algorithm

$$c = \frac{n * \sum_{i=0}^{n-1} a_i^2 - (\sum_{i=0}^{n-1} a_i)^2}{n(n-1)}$$

Domain

−3.4 x 10³⁸ to +3.4 x 10³⁸ for varf ()

−1.0 to +1.0 for var_fr16 ()

zero_cross

count zero crossing

Synopsis

```
#include <stats.h>
int zero_crossf(a,n)
const float a[];          /* Pointer to input vector a */
int n;                    /* Number of input samples */
int zero_cross_fr16 (a, n)
const fract16 a[];        /* Pointer to input vector a */
int n;                    /* Number of input samples */
```

Description

This function computes the number of times that a signal crosses over the zero line and returns the result.

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, this example should give you 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

-3.4×10^{38} to $+3.4 \times 10^{38}$ for zero_crossf ()

-1.0 to +1.0 for zero_cross_fr16 ()

I INDEX

Symbols

.IDL file [1-109](#)

@ filename (command file) compiler
switch [1-21](#)

__builtin prefix [1-76](#)

_primIO()

C function
[2-12](#)

_Sbrk() library function [1-99](#)

μ -law compression function [3-87](#)

μ -law expansion function [3-88](#)

Numerics

profiling

-p [1-96](#)

-21532 (compile for ADSP-21532)
compiler switch [1-21](#)

-21535 (compile for ADSP-21535)
compiler switch [1-21](#)

2-d convolution (conv2d) function [3-48](#)

2-d convolution (conv2d3x3) function
[3-50](#)

A

-A (assert) compiler switch [1-21](#)

a_compress (A-law compression)
function [3-26](#)

a_expand (A-law expansion) function
[3-27](#)

Abend (See abort function)

abort (abnormal program end) function
[2-24](#)

Abridged C++ library [2-13...2-19](#)

abs (absolute value) function [2-25](#)

accumulator register [1-80](#)

accumulators [1-105](#)

acos (arc cosine) function [2-26](#)

aggregate initializers [1-72](#)

aggregate return pointer [1-117](#)

A-law compression [3-26](#)

A-law expansion [3-27](#)

algebraic functions (See math functions)

ALIGN NUM pragma [1-86](#)

alloca() function [1-84](#)

allocate memory (See calloc, free,
malloc, realloc functions)

allocated events [1-105](#)

alphanumeric character test (See
isalnum function)

alternate keywords [1-30](#)

-alttok (alternative tokens) switch [1-22](#)

-analog (Analog C compilation)
compiler switch [1-20](#)

ANSI C signal handler [1-102](#)

INDEX

ANSI standard warnings [1-36](#)
ANSI/ISO standard C++ [1-20](#)
Application Binary Interface (ABI) [1-50](#)
arg (get phase of a complex number) function [3-28](#)
argument passing [1-121](#)
arguments and return transfer [1-121](#)
argv/argc
 arguments [1-93](#)
 support for [1-93](#)
argv/argc arguments [1-93](#)
arithmetic functions [3-4](#)
array search, binary (See bsearch function)
array sorting [2-75](#)
ASCII string (See atof, atoi, atol functions)
asin (arc sine) function [2-27](#)
asm()
 compiler keyword [1-56](#)
 (See also Assembly language support keyword (asm))
 constructs
 registers for [1-63](#)
 reordering [1-69](#)
 operand constraints [1-63](#), [1-66](#)
 template in C programs [1-59](#)
asm() construct
 operands [1-62](#)
asm() constructs
 syntax [1-59](#)
 syntax rules [1-60](#)
assembler (easmbldfn) [1-2](#)

assembly language support keyword (asm) [1-58](#)
atan (arc tangent) function [2-28](#)
atan2 (arc tangent of quotient) function [2-29](#)
atexit (select exit function) function [2-30](#)
atof (convert string to double) function [2-31](#)
atoi (convert string to integer) function [2-32](#)
atol (convert string to long integer) function [2-33](#)
autocoh (autocoherence) function [3-29](#)
autocoherence [3-29](#)
autocorr (autocorrelation) function [3-30](#)
autocorrelation [3-30](#)
-auto-inline (auto inline) compiler switch [1-22](#)

B

Bartlett window [3-64](#)
base 10 logarithms [2-64](#)
basic startup code [1-126](#)
binary array search (See bsearch function)
-BLACKFIN (Blackfin compiler) switch [1-23](#)
Blackfin DSP functionality [1-93](#)
 argv/argc arguments [1-93](#)
 default startup code [1-93](#)
 heap size [1-98](#)

- interrupts [1-99](#)
- MEM_ARGV section [1-93](#)
- profiling [1-95](#)
- Blackman window [3-65](#)
- Blackman-Harris window [3-69](#)
- blank space character [2-57](#)
- Boolean type support keywords
(bool, true, false) [1-71](#)
- bsearch (binary search in sorted
array) function [2-34](#)
- build-lib (build library) compiler
switch [1-23](#)
- built-in functions [1-76](#), [2-4](#)
- circular buffer [1-83](#)
- compiler [2-4](#)
- complex fract [1-79](#)
- fract16 [1-77](#)
- fract32 [1-77](#)
- fractional arithmetic [1-76](#)
- idle mode [1-85](#)
- IMASK [1-84](#)
- interrupts and exceptions [1-84](#)
- non-fractional [1-82](#)
- synchronization [1-85](#)
- system [1-84](#)
- system register read/write [1-84](#)
- Viterbi functions [1-80](#)

C

- C (comments) compiler switch
[1-23](#)
- c (compile only) compiler switch
[1-24](#)
- C and C++ library files [2-4](#)

- C data types [1-47](#)
- C function
_primIO() [2-12](#)
- C language extensions
 - C++ style comments [1-56](#)
 - indexed initializers [1-56](#)
 - non-constant initializers [1-56](#)
 - preprocessor-generated warnings
[1-56](#)
 - variable length arrays [1-56](#)
- C run-time
 - library guide [2-3](#)
 - library reference [2-23...2-119](#)
- C++ library
 - abridged [2-13...2-19](#)
- C++ mode compiler switches
 - explicit (explicit specifier) [1-43](#)
 - namespace [1-44](#)
 - newforinit (new for initialization)
[1-44](#)
 - newvec (new vector) [1-44](#)
 - no-demangle (disable demangler)
[1-44](#)
 - no-explicit (disable explicit
specifier) [1-44](#)
 - no-namespace [1-44](#)
 - no-newvec (disallow a new
vector) [1-45](#)
 - notstrict (non-strict compilation)
[1-45](#)
 - no-wchar (disable wide char type)
[1-45](#)
 - strict (strict standard) [1-45](#)
 - strictwarn [1-45](#)

INDEX

- tpautooff (disable automatic template instantiation) 1-46
- trdforinit (traditional initialization) 1-46
- typename 1-46
- wchar (enable wide char type) 1-46
- C++ programming examples 1-133
 - complex support 1-134
 - fract support 1-133
- C++ style comments 1-75
- C/assembly interfacing (See mixed C/assembly programming)
- C/C++ compiler
 - guide 1-1, 1-2
 - overview 1-1, 1-2
- C/C++ compiler mode switches
 - analog 1-20
 - c++ (C++ mode) 1-20
 - traditional (traditional compilation) 1-20
- C/C++ language extensions 1-55...1-92
 - asm keyword 1-58
 - bool keyword 1-56
 - false keyword 1-56
 - inline keyword 1-57
 - restrict 1-56
 - section keyword 1-56
 - true keyword 1-56
- C/C++ run-time environment
 - (See also mixed C/C++/assembly programming)
 - environment, defined 1-113
 - library guide 2-3...2-19
- cabs (complex absolute value) function 3-31
- cadd (complex addition) function 3-32
- call preserved registers 1-116
- calling assembly language subroutine 1-128
- calling library functions 2-3
- calloc (allocate and initialize memory) function 2-36
- ccblkfn (Blackfin C/C++ compiler) 1-1, 1-2
- cdiv (complex division) function 3-33
- ceil (ceiling) function 2-37
- cexp (complex exponential) function 3-34
- cfft (n point radix 2 complex fft) function 3-35
- cfft2d (n x n point 2-d complex input fft) function 3-39
- cfftrad4 (n point radix 4 complex fft) function 3-37
- cfir (complex FIR filter) function 3-41
- changing memory allocation 2-80
- char storage format 1-124
- character string search (See strchr function)
- circbuf (circular buffer) compiler switch 1-24
- circular buffer registers 1-105

- circular buffers [1-116](#)
 - built-ins [1-83](#)
- clip (clip) function [3-43](#)
- clobber string specifiers [1-66](#)
- cmlt (complex multiply) function [3-44](#)
- command-line
 - interface [1-4...1-52](#)
 - syntax [1-5](#)
- comments [1-75](#)
- comparing characters in strings [2-96](#)
- comparing null-terminated strings [2-89](#)
- compiler
 - built-in functions [2-4](#)
 - C/C++ language extensions [1-55](#)
 - command-line syntax [1-5](#)
- compiler common switches
 - @ filename [1-21](#)
 - 21532 (compile for ADSP-21532) [1-21](#)
 - 21535 (compile for ADSP-21535) [1-21](#)
 - A (assert) [1-21](#)
 - alttok (alternative tokens) [1-22](#)
 - auto-inline (auto inline) [1-22](#)
 - BLACKFIN (compile) [1-23](#)
 - build-lib (build library) [1-23](#)
 - C (comments) [1-23](#)
 - c (compile only) [1-24](#)
 - circbuf [1-24](#)
 - D (define macro) [1-25](#)
 - debug-types [1-24](#)
 - dry (a verbose dry-run) [1-25](#)
 - dryrun (a terse dry-run) [1-25](#)
 - E (stop after preprocessing) [1-25](#)
 - EE (run after preprocessing) [1-26](#)
 - extra-keywords (enable short-form keywords) [1-26](#)
 - flags (command-line input) [1-26](#)
 - full-version (display version) [1-26](#)
 - g (generate debug information) [1-26](#)
 - H (list headers) [1-27](#)
 - help (command-line help) [1-27](#)
 - HH (list headers and compile) [1-27](#)
 - I (include search directory) [1-28](#)
 - include (include file) [1-28](#)
 - ipa [1-28](#)
 - jcs21 [1-28](#)
 - L (library search directory) [1-28](#)
 - l (link library) [1-28](#)
 - M (generate make rules only) [1-29](#)
 - map (generate a memory map) [1-30](#)
 - MM (generate make rules and compile) [1-29](#)
 - Mt (output make rule for named file) [1-29](#)
 - no-alttok (disable alternative tokens) [1-30](#)
 - no-builtin (no built-in functions) [1-30](#)
 - no-defs (disable defaults) [1-30](#)
 - no-extra-keywords [1-30](#)

INDEX

- no-inline (disable inline keyword) [1-31](#)
- no-int-to-fact (disable integer to fractional conversion) [1-31](#)
- no-int-to-fract [1-31](#)
- no-restrict (disable restrict) [1-31](#)
- no-std-def (disable standard macro definitions) [1-31](#)
- no-std-inc (disable standard include search) [1-32](#)
- no-std-lib (disable standard library search) [1-32](#)
- nothreads (disable thread-safe build) [1-32](#)
- O (enable optimizations) [1-32](#)
- o (output file) [1-34](#)
- Ofp (frame pointer optimizations) [1-33](#)
- Os (enable code size optimizations) [1-33](#)
- Ox (sign-extension optimizations) [1-33](#), [1-50](#)
- Oz (low program memory) [1-34](#)
- p (generate profiling implementation) [1-35](#)
- P omit line numbers) [1-34](#)
- path (tool location) [1-35](#)
- path-install (installation location) [1-35](#)
- path-output (non-temporary files location) [1-35](#)
- path-temp (temporary files location) [1-36](#)
- pedantic (ANSI standard warnings) [1-36](#)
- pedantic-errors (ANSI standard errors) [1-36](#)
- PP (omit line numbers and run) [1-34](#)
- pplist (preprocessor listing) [1-36](#)
- proc processorID [1-37](#)
- R (add source directory) [1-37](#)
- ref (cross-reference list) [1-42](#)
- restrict [1-37](#)
- S (stop after compilation) [1-38](#)
- s (strip debug information) [1-38](#)
- sat32 (32 bit saturation) [1-38](#)
- sat40 40 bit saturation) [1-38](#)
- save-temps (save intermediate files) [1-38](#)
- show (display command line) [1-39](#)
- signed-char (make char signed) [1-39](#)
- sourcefile [1-21](#)
- syntax-only (just check syntax) [1-39](#)
- T (linker description file) [1-39](#)
- threads (enable thread-safe build) [1-39](#)
- time (tell time) [1-40](#)
- U (undefine macro) [1-40](#)
- unsigned-char (make char unsigned) [1-40](#)
- v (version and verbose) [1-40](#)
- verbose (display command line) [1-40](#)
- version (display version) [1-40](#)

- w (disable all warnings) [1-42](#)
- W (override error message) [1-41](#)
- warn-protos (prototypes warning) [1-41](#)
- Wdriver-limit (maximum process errors) [1-41](#)
- Werror-limit (maximum compiler errors) [1-41](#)
- Wremarks (enable diagnostic warnings) [1-41](#)
- write-files (enable driver I/O redirection) [1-42](#)
- Wterse (enable terse warnings) [1-42](#)
- complex
 - absolute value [3-31](#)
 - addition [3-32](#)
 - conjugate [3-45](#)
 - division [3-33](#)
 - exponential [3-34](#)
 - fract built-ins [1-79](#)
 - functions [3-4](#)
 - multiply [3-44](#)
 - number [3-28](#)
 - subtraction [3-57](#)
- complex.h header file [3-4](#)
- complex_fract16 [1-79](#)
- complex_fract32 [1-79](#)
- compression/expansion [3-9](#)
- condition code register (CC) [1-105](#)
- conj (complex conjugate) function [3-45](#)
- const-read-write [1-24](#)
- constructs
 - flow control [1-70](#)
 - input and output operands [1-69](#)
 - operand description [1-62](#)
 - reordering and optimization [1-68](#)
 - template for assembly [1-59](#)
 - with multiple instructions [1-68](#)
- control character test (See isctrl function)
- controlling
 - inlining [1-51](#)
 - optimization [1-48](#)
- conv2d (2-d convolution) function [3-48](#)
- conv2d3x3 (2-d convolution) function [3-50](#)
- conventions, of this manual [-xxxi](#)
- convert
 - characters (See tolower, toupper functions)
 - strings (See atof, atoi, atol, strtok, strtol, strtoul, functions)
- convolve (convolution) function [3-46](#)
- convolve transformations [3-4](#)
- copying characters from one string to another [2-97](#)
- copying from one string to another [2-91](#)
- copysign (copysign) function [3-51](#)
- cos (cosine) function [2-38](#)
- cosh (hyperbolic cosine) function [2-39](#)
- Cosine window [3-68](#)
- cot (cotangent) function [3-52](#)

INDEX

cotangent 3-52
counting one bits in word 3-53
countones (count one bits in word)
 function 3-53
crosscoh (cross-coherence) function
 3-54
crosscorr (cross-correlation)
 function 3-56
csub (complex subtraction) function
 3-57
C-type functions
 iscntrl 2-51
 isgraph 2-53
 islower 2-54
 isprint 2-55
 ispunct 2-56
 isspace 2-57
 isupper 2-58
 isxdigit 2-59
 tolower 2-114
 toupper 2-115
customer support -xxv
cycle counter 1-98

D

-D (define macro) compiler switch
 1-25, 1-40
data alignment pragmas 1-86
data storage formats
 using 1-124
data type (storage) sizes 1-124
data types 1-47

 sizes 1-47
deallocate memory (See free
 function)
debug information
 removing 1-38
-debug-types compiler switch 1-24
dedicated registers 1-116
default
 event handlers 1-105
 startup code 1-105
default LDF 1-98
demangler 1-44
detect punctuation character
 (ispunct) function 2-56
detecting control character 2-51
div (division) function 2-40
division
 complex 3-33
division (See div, ldiv functions)
DIVS and DIVQ functions 1-82
double data type 1-48
double representation 2-102
double storage format 1-124
-dry (terse -dry-run) compiler
 switch 1-25
-dry-run (verbose dry-run) compiler
 switch 1-25
DSP
 filters 3-4
 header files 3-3
 library functions, linking 3-2
 run-time library format 3-25

E

- E (stop after preprocessing)
 - compiler switch [1-25](#)
- EE (run after preprocessing)
 - compiler switch [1-26](#)
- embedded C++ header files
 - `complex.h` [2-13](#)
 - `exception.h` [2-13](#)
 - `fstream.h` [2-14](#)
 - `iomanip.h` [2-14](#)
 - `ios.h` [2-14](#)
 - `iosfwd.h` [2-14](#)
 - `iostream.h` [2-14](#)
 - `istream.h` [2-14](#)
 - `new.h` [2-14](#)
 - `ostream.h` [2-15](#)
 - `sstream.h` [2-15](#)
 - `stdexcept.h` [2-15](#)
 - `streambuf.h` [2-15](#)
 - `string.h` [2-15](#)
 - `strstream.h` [2-15](#)
- Embedded C++ Library [2-13](#)
- embedded standard template library
 - [2-17](#)
- End (See `atexit`, `exit` functions)
- `errno.h` header file [2-9](#)
- ETSI_SOURCE fract functions
 - [1-78](#)
- event context [1-103](#)
- event details
 - exceptions [1-103](#)
 - fetching [1-103](#)
- event vector table [1-101](#), [1-102](#)
- events

- allocated values in user mode
 - [1-105](#)
- EX_EXCEPTION_HANDLER
 - compiler macro [1-100](#)
- EX_INT_DEFAULT value [1-101](#)
- EX_INT_IGNORE value [1-101](#)
- EX_INTERRUPT_HANDLER
 - compiler macro [1-100](#)
- EX_NMI_HANDLER compiler
 - macro [1-100](#)
- EX_REENTRANT_HANDLER
 - compiler macro [1-100](#)
- EXCAUSE values [1-103](#), [1-105](#)
- executable
 - running [1-97](#)
- exit (normal program termination)
 - function [2-41](#)
- exp (exponential) function [2-42](#)
- explicit (explicit specifier) C++
 - mode compiler switch [1-43](#)
- extra-keywords (not quite -analog)
 - compiler switch [1-26](#)

F

- fabs (float absolute value) function
 - [2-43](#)
- false (See Boolean type support
 - keywords (`bool`, `true`, `false`))
- far jump return (See `longjmp`,
 - `setjmp` functions)
- Fast Fourier Transforms [3-4](#), [3-7](#)
- `fclose()` function [2-12](#)
- fetching event details [1-103](#)
- FFT function versions [3-6](#)

INDEX

- file extensions [1-5](#), [1-7](#), [1-21](#)
- file I/O access [1-94](#)
- file searching [1-7](#)
 - <filename> [1-109](#)
- filenames [1-21](#)
- filter library [3-6](#)
- filter.h header file [3-4](#)
- filters
 - in digital signal processing [3-4](#)
 - library [3-6](#)
- FIR (finite impulse response filter)
 - function [3-58](#)
- fir_decima (fir decimation filter)
 - function [3-60](#)
- fir_interp (FIR interpolation filter)
 - function [3-62](#)
- flags (command line input)
 - compiler switch [1-26](#)
- float storage format [1-124](#)
- float.h (floating point) header file [2-9](#)
- floor () function [2-44](#)
- FLT_ROUNDS macro [2-9](#)
- fmod (floating-point modulus)
 - function [2-45](#)
- fopen() function [2-11](#)
- fprintf() function [2-11](#)
- fract.h header file [1-77](#), [1-78](#)
- fract16 type [1-77](#)
- fract32 [1-77](#)
- fractional literals [1-79](#)
- fractional value
 - built-ins [1-76](#)
 - C types [1-76](#)
 - complex built-ins [1-79](#)
- frame pointer [1-50](#), [1-104](#), [1-117](#), [1-118](#), [1-119](#)
 - dedicated register [1-116](#)
- fread() function [2-12](#)
- free (deallocate memory) function [2-46](#)
- frexp (separate fraction and exponent) function [2-47](#)
- fstreams.h header file [2-18](#)
- full-version (display version)
 - compiler switch [1-26](#)
- functions
 - arguments/return value transfer [1-121](#)
 - arithmetic [3-4](#)
 - complex [3-4](#)
 - DIVS and DIVQ [1-82](#)
 - entry (prologue) [1-117](#)
 - exit (epilogue) [1-117](#)
 - math [3-10](#)
 - matrix [3-12](#)
 - primitive I/O [2-11](#)
 - statistical [3-16](#)
 - transformational [3-6](#)
 - vector [3-18](#)
- fwrite() function [2-12](#)
- G**
 - g (generate debug information)
 - compiler switch [1-26](#)
 - Gaussian window [3-66](#)
 - gen_bartlett (generate bartlett window) function [3-64](#)

- gen_blackman (generate blackman window) function [3-65](#)
- gen_gaussian (generate gaussian window) function [3-66](#)
- gen_hamming (generate hamming window) function [3-67](#)
- gen_hanning (generate hanning window) function [3-68](#)
- gen_harris (generate harris window) function [3-69](#)
- gen_kaiser (generate kaiser window) function [3-70](#)
- gen_rectangular (generate rectangular window) function [3-71](#)
- gen_triangle (generate triangle window) function [3-72](#)
- gen_vonhann (generate von hann window) function [3-74](#)
- general optimization pragmas [1-89](#)
- generating instrumented code [1-96](#)
- get_interrupt_info() function [1-103](#)
- getting string containing error message [2-93](#)
- global variable [1-132](#)
- graphical character test (See isgraph function)

H

- H (list *.h) compiler switch [1-27](#)
- Hamming window [3-67](#)
- Hanning window [3-68](#)
- hardware error value [1-104](#)
- hardware loop registers [1-105](#)

- Harris window [3-69](#)
- header files [3-3](#)
 - accessing [1-110](#)
 - C/C++ declarations [1-110](#)
 - macro definitions [1-110](#)
 - search for [1-32](#)
 - standard C library [2-8...??](#)
- header files (new form)
 - cassert.h [2-16](#)
 - cctype.h [2-16](#)
 - cerrno.h [2-16](#)
 - cfloat.h [2-16](#)
 - climits.h [2-16](#)
 - locale.h [2-16](#)
 - cmath.h [2-16](#)
 - csetjmp.h [2-16](#)
 - csignal.h [2-16](#)
 - cstdarg.h [2-16](#)
 - cstddef.h [2-16](#)
 - cstdio.h [2-16](#)
 - cstdlib.h [2-16](#)
 - cstring.h [2-16](#)
- header files (standard)
 - assert.h [2-9](#)
 - ctype.h [2-9](#)
 - errno.h [2-9](#)
 - float.h [2-9](#)
 - limits.h [2-10](#)
 - locale.h [2-10](#)
 - math.h [2-10](#)
 - setjmp.h [2-11](#)
 - signal.h [2-11](#)
 - stdarg.h [2-11](#)
 - stddef.h [2-11](#)

INDEX

- stdio.h [2-11](#)
- stdlib.h [2-12](#)
- string.h [2-12](#)
- header files (template library)
 - <algorithm> [2-17](#)
 - <deque> [2-17](#)
 - <functional> [2-17](#)
 - <hash_map> [2-17](#)
 - <hash_set> [2-17](#)
 - <iterator> [2-17](#)
 - <list> [2-17](#)
 - <map> [2-17](#)
 - <memory> [2-18](#)
 - <numeric> [2-18](#)
 - <queue> [2-18](#)
 - <set> [2-18](#)
 - <stack> [2-18](#)
 - <utility> [2-18](#)
 - <vector> [2-18](#)
- heap [1-99](#)
 - memory control [1-98](#)
 - section [1-115](#)
- help (command-line help)
 - compiler switch [1-27](#)
- hexadecimal digit test (See isxdigit function)
- HH (list *.h and compile) compiler switch [1-27](#)
- histogram (histogram) function [3-75](#)
- hyperbolic sine function [2-84](#)
- hyperbolic tangent [2-113](#)

I

- I (include search directory)
 - compiler switch [1-28](#), [1-32](#)
- I/O interface [2-11](#)
- IEEE single/double precision
 - description [1-124](#)
- ifft (n point radix 2 inverse fft)
 - function [3-76](#)
- ifft2d (n x n point 2-d inverse input fft) function [3-80](#)
- ifft4 (n point radix 4 inverse input fft) function [3-78](#)
- iir function [3-82](#)
- include (include file) compiler switch [1-28](#)
- include directives [1-109](#)
- indexed initializer support [1-73](#)
 - compiler [1-73](#)
- infinite impulse response filter [3-82](#)
- initializers
 - aggregate [1-72](#)
- inline assembly language support
 - keyword (asm) [1-58](#)
- constructs
 - optimization [1-69](#)
 - template [1-59](#)
 - template operands [1-63](#)
- constructs with multiple instructions [1-68](#)
- inline function support keyword (inline) [1-56](#), [1-57](#)
- inlining [1-51](#)
- installation location [1-35](#)

- install (instantiate all) C++
mode compiler switch [1-43](#)
- instantiate [1-43](#)
- instantused (instantiate used) C++
mode compiler switch [1-43](#)
- int storage format [1-124](#)
- interfacing C/C++ and assembly
(See mixed C/C++/assembly programming)
- intermediate files [1-38](#)
- interrupt
function [2-48](#)
- interrupt handler pragmas [1-88](#)
- interrupt service routines (ISR) [1-99](#)
- ioanip.h header file [2-18](#)
- iostream.h header file [2-19](#)
- ipa compiler switch [1-28](#)
- isalnum (detect alphanumeric character) function [2-49](#)
- isalpha (detect alphabetic character) function [2-50](#)
- isctrl (detect control character) function [2-51](#)
- isdigit (detect decimal digit) function [2-52](#)
- isgraph (detect printable character) function [2-53](#)
- isgraph function [2-53](#)
- islower (detect lowercase character) function [2-54](#)
- isprint (detect printable character) function [2-55](#)
- ispunct (detect punctuation character) function [2-56](#)

- ISR
saved registers [1-104](#)
- ISR function
registering [1-101](#)
- isspace (detect whitespace character) function [2-57](#)
- isupper (detect uppercase character) function [2-58](#)
- isxdigit (detect hexadecimal digit) function [2-59](#)

J

- jcs2l compiler switch [1-28](#)

K

- Kaiser window [3-70](#)
- keywords (compiler)
(See also compiler C/C++ extensions)
extensions [1-30](#)

L

- L (library search directory)
compiler switch [1-28](#)
- l (link library) compiler switch
[1-28](#), [1-32](#)
- L_mac() macro [1-79](#)
- L_msu() macro [1-79](#)
- labs (long integer absolute value) function [2-60](#)
- language extensions (compiler) (See compiler C/C++ extensions)
- LC_COLLATE locale category
[2-110](#)

INDEX

- ldexp (multiply by power of 2)
 - function [2-61](#)
- ldiv (division) function [2-62](#)
- library
 - C run-time guide [2-3](#)
 - C run-time reference [2-23...2-119](#)
 - C++ abridged [2-13...2-19](#)
 - calling functions [2-3](#)
 - format for DSP run-time [3-25](#)
 - linking DSP functions [3-2](#)
 - linking functions [2-4](#)
 - source code
 - working with [3-2](#)
- library functions [2-20](#)
- limits.h header file [2-10](#)
- linkage_name pragma [1-90](#)
- Linker Description File (LDF) [1-39](#)
- linking
 - DSP library functions [3-2](#)
 - explicitly with the linker [1-96](#)
 - library functions [2-4](#)
 - pragmas [1-90](#)
 - the executable for profiling [1-96](#)
 - with ccbkfn [1-96](#)
- locale.h header file [2-10](#)
- log (natural logarithm) function [2-63](#)
- log10 (base 10 logarithm) function [2-64](#)
- long file names
 - handling with -write-files switch [1-42](#)
- long int storage format [1-124](#)

- long jump (See longjmp, setjmp functions)
- longjmp (second return from setjmp) function [2-65](#)
- loop
 - optimization [1-88](#)
 - pragmas [1-88](#)
 - vectorizing [1-90](#)
- lower case (See islower, tolower functions)

M

- M (make only) compiler switch [1-29](#)
- macros
 - compound statements [1-110](#)
 - HUGE_VAL [2-10](#)
 - interrupt handler [1-100](#)
 - predefined [1-107](#)
 - writing [1-110](#)
- malloc (allocate memory) function [2-67](#)
- map (generate a memory map)
 - compiler switch [1-30](#)
- map file [1-30](#)
- math functions [3-10](#)
 - library [3-10](#)
- math.h (mathematics) header file [2-10](#)
- math.h header file [3-10](#)
- matrix functions [3-12](#)
- matrix.h header file [3-12](#)
- max (maximum) function [3-84](#)
- MDUSERMODE flag [1-105](#)

- mean (mean) function [3-85](#)
 - memchr (find first occurrence of character) function [2-68](#)
 - memcmp (compare objects) function [2-69](#)
 - memcpy (copy characters from one object to another) function [2-70](#)
 - memmove (move characters from one object to another) function [2-71](#)
 - memory
 - allocation functions [1-115](#)
 - map [1-98](#)
 - using memory sections [1-114](#)
 - memory (See calloc, free, malloc, memcmp, memcpy, memset, memmove, memchar, realloc functions)
 - memory size
 - controlling [1-98](#)
 - memset (set range of memory to a character) function [2-72](#)
 - min (minimum) function [3-86](#)
 - mixed C/Assembly naming
 - conventions [1-131](#)
 - mixed C/assembly programming
 - arguments and return [1-121](#)
 - asm() constructs [1-58](#), [1-59](#), [1-63](#), [1-68](#), [1-69](#)
 - data storage and type sizes [1-124](#)
 - scratch registers [1-117](#)
 - stack registers [1-117](#)
 - stack usage [1-117](#)
 - mixed C/assembly reference [1-130](#)
 - mixed C/C++/assembly
 - programming [1-113](#)
 - mixed C/C++/assembly reference [1-113](#)
 - MM (make and compile) compiler switch [1-29](#)
 - mode selection switches [1-10](#)
 - modf (separate integral and fractional parts) function [2-73](#)
 - mon.out file [1-97](#)
 - move memory range (See memmove function)
 - Mt preprocessor switch [1-29](#)
 - mu_compress (μ -law compression) function [3-87](#)
 - mu_expand (μ -law expansion) function [3-88](#)
 - multiline asm() C program constructs [1-68](#)
- ## N
- namespace (enable namespaces) C++ mode compiler switch [1-44](#)
 - naming conventions
 - C and assembly [1-132](#)
 - natural logarithms [2-63](#)
 - new.h header file [2-19](#)
 - newforinitl (new for initialization) C++ mode compiler switch [1-44](#)
 - newvec (new vector) C++ mode compiler switch [1-44](#)

INDEX

- next argument in variable list [2-116](#)
- NMI events [1-104](#)
- no_alias pragma [1-89](#), [1-92](#)
- no-alttok (disable tokens) C++
mode compiler switch [1-30](#)
- no-builtin (no builtin functions)
compiler switch [1-30](#)
- no-char (disable wide char type)
C++ mode compiler switch
[1-45](#)
- no-def (disable definitions)
compiler switch [1-30](#)
- no-demangle (disable demangler)
C++ mode compiler switch
[1-44](#)
- no-explicit (disable explicit
specifier) C++ mode compiler
switch [1-44](#)
- no-extra-keywords (not quite -ansi)
compiler switch [1-30](#)
- no-inline (disable inline keyword)
compiler switch [1-31](#)
- no-int-to-fract (disable integer to
fractional conversion) compiler
switch [1-31](#)
- no-jcs2l compiler switch [1-31](#)
- no-namespace (disable
namespaces) C++ mode
compiler switch [1-44](#)
- Non-constant initializer support
(compiler) [1-72](#)
- non-constant initializer support
(compiler) [1-72](#), [1-75](#)
- no-newvec (disallow a new vector)
C++ mode compiler switch
[1-45](#)
- non-fractional built-in functions
[1-82](#)
- non-fractional built-ins [1-82](#)
- no-restrict (no restrict support)
C++ mode compiler switch
[1-31](#)
- norm (normalization) function [3-89](#)
- no-std-def (disable standard
definitions) compiler switch
[1-31](#)
- no-std-inc (disable standard
include search) compiler switch
[1-32](#)
- no-std-lib (disable standard library
search) compiler switch [1-32](#)
- nothreads (disable thread-safe
build) compiler switch [1-32](#)
- notstrict (non-strict compilation)
C++ mode compiler switch
[1-45](#)
- O**
- O (enable optimization) compiler
switch [1-32](#), [1-33](#)
- o (output) compiler switch [1-34](#)
- offsetting frame pointer [1-50](#)
- Ofp (frame pointer optimizations)
switch [1-33](#)
- operand constraints [1-64](#)
- optimization control features [1-48](#)
- optimization switches [1-49](#), [1-50](#)

optimize_for_space pragma 1-89
 optimize_for_speed pragma 1-89
 optimize_off pragma 1-89
 optimizer 1-90
 optimizing asm() C program
 constructs 1-69
 -Ox (sign-extension optimizations)
 switch 1-33, 1-50
 -Oz (low program memory) switch
 1-34

P

-P (omit #line) compiler switch
 1-34, 1-35
 PACK (ALIGNOPT) pragma 1-87
 PAD (ALIGNOPT) pragma 1-87
 parameter passing 1-121
 passing arguments 1-121
 -path-install (installation location)
 compiler switch 1-35
 -path-output (non-temporary files
 location) compiler switch 1-35
 -path-temp (temporary files
 location) compiler switch 1-36
 -path-tool (tool location) compiler
 switch 1-35
 -pedantic (ANSI standard warnings)
 compiler switch 1-36
 -pedantic-errors (ANSI standard
 errors) compiler switch 1-36
 perror (map error number to error
 message) function 2-74
 placement support keyword
 (section) 1-70

pointer class support keyword
 (restrict) 1-56, 1-71
 polar (construct from polar
 coordinates) function 3-90
 post-processing mon.out file from
 profiler 1-97
 pow (raise to a power) function 2-74
 pow function 2-74
 -pplist (preprocessor listing)
 compiler switch 1-36
 pragmas 1-85, 1-91
 ALIGN NUM 1-86
 data alignment 1-86
 exception 1-88
 interrupt 1-88
 linkage_name 1-90
 linking 1-90
 loop 1-88
 nmi 1-88
 no_alias 1-89, 1-91
 optimization 1-89
 optimize_for_space 1-89
 optimize_for_speed 1-89
 optimize_off 1-89
 pack() 1-87
 pad () 1-87
 retain_name 1-90
 VECTOR_FOR 1-89
 predefined macros 1-107
 __ADSPBLACKFIN__ 1-108
 __ANALOG_EXTENSIONS__
 1-108
 __cplusplus 1-108
 __DATE__ 1-108

INDEX

- [__ECC__ 1-108](#)
- [__EDG__ 1-108](#)
- [__EDG_VERSION__ 1-108](#)
- [__FILE__ 1-108](#)
- [__LINE__ 1-108](#)
- [__NO_BUILTIN 1-108](#)
- [__NO_LONGLONG 1-108](#)
- [__STDC__ 1-108](#)
- [__STDC_VERSION__ 1-109](#)
- [__TIME__ 1-109](#)
- [ADSPBLACKFIN 1-108](#)
- [prelinker 1-43, 1-52](#)
- [preprocessing
 - \[a program 1-107\]\(#\)
 - \[IDL files 1-109\]\(#\)](#)
- [preprocessor
 - \[generated warnings 1-75\]\(#\)](#)
- [preserved registers 1-116](#)
- [primitive I/O functions 2-11](#)
- [printable character test \(See isprint function\)](#)
- [printable characters 2-53, 2-55](#)
- [printf\(\) function 2-11](#)
- [-proc \(target processor\) compiler
 - \[switch 1-37\]\(#\)](#)
- [processor context on supervisor
 - \[stack 1-102\]\(#\)](#)
- [proflbkfn.exe program 1-97](#)
- [profiling
 - \[executable outputs 1-97\]\(#\)
 - \[library 1-98\]\(#\)
 - \[routine 1-95\]\(#\)
 - \[with linking 1-96\]\(#\)](#)
- [program control functions](#)

- [calloc 2-36](#)
- [malloc 2-67](#)
- [realloc 2-80](#)

Q

- [qsort \(quicksort\) function 2-75](#)

R

- [-R \(search for source files\) compiler
 - \[switch 1-37\]\(#\)](#)
- [raise \(raise a signal\) function 2-77](#)
- [rand \(random number generator\) function 2-79](#)
- [random number generator \(See rand, srand functions\) 2-79](#)
- [realloc \(change memory allocation\) function 2-80](#)
- [reciprocal square root \(rsqrt\) function 3-98](#)
- [rectangular window 3-71](#)
- [registers
 - \[call preserved 1-116\]\(#\)
 - \[dedicated 1-116\]\(#\)
 - \[for asm\\(\\)
 - \\[constructs 1-63\\]\\(#\\)\]\(#\)
 - \[preserved 1-116\]\(#\)
 - \[saved during ISR prologue 1-104\]\(#\)
 - \[scratch 1-117\]\(#\)
 - \[stack 1-117\]\(#\)
 - \[usage \\(See mixed C/assembly programming\\)\]\(#\)](#)
- [restrict \(See pointer class support keyword \(restrict\)\)](#)

- restrict (support restrict keyword)
C++ mode compiler switch
[1-37](#)
- retain_name pragma [1-90](#)
- return value transfer [1-121](#)
- return values [1-122](#)
- returning
 - floating-point input multiplied by
2 [2-61](#)
 - long integer absolute value [2-60](#)
- rfft (n point radix 2 real input fft)
function [3-91](#)
- rfft2d (n x n point 2-d real input fft)
function [3-95](#)
- rfftr4d (n point radix 4 real input
fft) function [3-93](#)
- rms (root mean square) function
[3-97](#)
- root mean square (rms) function
[3-97](#)
- rsqrt (reciprocal square root)
function [3-98](#)
- running the executable [1-97](#)
- run-time environment [1-113](#)
(See also mixed C/C++/assembly
programming)
programming (See mixed
C/C++/assembly
programming)
- run-time stack [1-117](#)
- S**
- S (stop after compilation) compiler
switch [1-38](#)
- s (strip debug information)
compiler switch [1-38](#)
- sat32 (32 bit saturation) compiler
switch [1-38](#)
- sat40 (40 bit saturation) compiler
switch [1-38](#)
- SAVE_REGS() compiler macro
[1-103](#)
- saved registers in SYSSTACK [1-104](#)
- save-temps (save intermediate files)
compiler switch [1-38](#)
- scratch registers [1-117](#)
- search
 - character string (See strchr, strrchr
functions)
 - memory, character (See memchar
function)
 - path for include files [1-28](#)
 - path for library files [1-28](#)
- searching for #included files [1-109](#)
- segment (See Placement support
keyword (section))
- sending signal to executing program
[2-77](#)
- set jump (See longjmp, setjmp
functions)
- setjmp (label for external linkage)
function [2-81](#)
- setjmp.h header file [2-11](#)
- setting range of memory to a
character [2-72](#)
- short storage format [1-124](#)
- show (display command line)
compiler switch [1-39](#)

INDEX

signal (define signal handling)
 function [2-82](#)
signal handler [1-102](#)
signal handling [2-82](#)
signal.h (signal handling) header file
 [2-11](#)
-signed-char (make char signed)
 compiler switch [1-39](#)
simulator [1-93](#), [1-94](#)
sin (sine) function [2-83](#)
single fractional values [1-77](#)
sinh (hyperbolic sine) function [2-84](#)
space allocator [1-84](#)
sqrt (square root) function [2-85](#)
srand (random number seed)
 function [2-86](#)
stack [1-117](#)
 managing [1-117](#)
 registers [1-117](#)
stack pointer [1-117](#)
stack pointer dedicated register
 [1-116](#)
Standard C Library [2-15](#)
Standard library functions
 abort [2-24](#)
 abs [2-25](#)
 acos [2-26](#)
 asin [2-27](#)
 atan [2-28](#)
 atan2 [2-29](#)
 atexit [2-30](#)
 atof [2-31](#)
 atoi [2-32](#)
 atol [2-33](#)
 bsearch [2-34](#)
 calloc [2-36](#)
 ceil [2-37](#)
 cos [2-38](#)
 cosh [2-39](#)
 div [2-40](#)
 exit [2-41](#)
 exp (exponential) [2-42](#)
 fabs [2-43](#)
 floor [2-44](#)
 fmod [2-45](#)
 free [2-46](#)
 frexp [2-47](#)
 isalnum [2-49](#)
 isalpha [2-50](#)
 iscntrl [2-51](#)
 isdigit [2-52](#)
 isgraph [2-53](#)
 islower [2-54](#)
 isprint [2-55](#)
 isspace [2-57](#)
 isupper [2-58](#)
 isxdigit [2-59](#)
 labs [2-60](#)
 ldexp [2-61](#)
 ldiv [2-62](#)
 log [2-63](#)
 log10 [2-64](#)
 longjmp [2-65](#)
 malloc [2-67](#)
 memchr [2-68](#)
 memcmp [2-69](#)
 memcpy [2-70](#)
 memmove [2-71](#)

- memset [2-72](#)
- modf [2-73](#)
- pow [2-74](#)
- qsort [2-75](#)
- raise [2-77](#)
- rand [2-79](#)
- realloc [2-80](#)
- setjmp [2-81](#)
- signal [2-82](#)
- sin [2-83](#)
- sqrt [2-85](#)
- srand [2-86](#)
- strbrk [2-98](#)
- strcat [2-87](#)
- strchr [2-88](#)
- strcmp [2-89](#)
- strcoll [2-90](#)
- strcpy [2-91](#)
- strcspn [2-92](#)
- strerror [2-93](#)
- strlen [2-94](#)
- strncat [2-95](#)
- strncmp [2-96](#)
- strncpy [2-97](#)
- strrchr [2-99](#)
- strspn [2-100](#)
- strstr [2-101](#)
- strtok [2-104](#)
- strtol [2-106](#)
- strtoul [2-108](#)
- strxfrm [2-110](#)
- tan [2-112](#)
- tanh [2-113](#)
- tolower [2-114](#)
- toupper [2-115](#)
- va_arg macro [2-116](#)
- va_end macro [2-118](#)
- statistical functions [3-16](#)
- stats.h header file [3-16](#)
- stdarg.h header file [2-11](#)
- stddef.h header file [2-11](#)
- stdio functions [1-94](#)
- stdio.h header file [2-11](#)
- stdlib.h header file [2-12](#)
- stop (See atexit, exit functions)
- storage formats
 - short [1-124](#)
- strcat (concatenate strings) function [2-87](#)
- strchr (find first occurrence of character in string) function [2-88](#)
- strcmp (compare strings) function [2-89](#)
- strcoll (compare strings) function [2-90](#)
- strcpy (copy from one string to another) function [2-91](#)
- strcspn (compare string span) function [2-92](#)
- strerror (get string containing error message) function [2-93](#)
- strerror function [2-93](#)
- strict (strict compilation) C++ mode compiler switch [1-45](#)
- strictwarn (warn if non-strict) C++ mode compiler switch [1-45](#)

INDEX

- string conversion (See `atof`, `atoi`,
 `atol`, `strtok`, `strtol`, `strxfrm`
 functions)
- string functions
 - `memchar` [2-68](#)
 - `memmove` [2-71](#)
 - `strchr` [2-88](#)
 - `strcoll` [2-90](#)
 - `strcspn` [2-92](#)
 - `strerror` [2-93](#)
 - `strpbrk` [2-98](#)
 - `strrchr` [2-99](#), [2-100](#)
 - `strspn` [2-100](#)
 - `strstr` [2-101](#)
 - `strtok` [2-104](#)
 - `strxfrm` [2-110](#)
- string transformation using
 `LC_COLLATE` [2-110](#)
- `string.h` header file [2-12](#)
- `strlen` (string length) function [2-94](#)
- `strncat` (concatenate characters from
 one string to another) function
 [2-95](#)
- `strncmp` (compare characters in
 strings) function [2-96](#)
- `strncpy` (copy characters from one
 string to another) function [2-97](#)
- `strpbrk` (find character match in two
 strings) function [2-98](#)
- `strrchr` (find last occurrence of
 character in string) function
 [2-99](#)
- `strspn` (length of segment of
 characters in both strings)
 function [2-100](#)
- `strstr` (compare string, string)
 function [2-101](#)
- `strstr` (find string within string)
 function [2-101](#)
- `strtod` (convert portion of string to
 double) function [2-102](#)
- `strtok` (convert string to tokens)
 function [2-104](#)
- `strtol` (convert string to long integer)
 function [2-106](#)
- `strtoul` (convert string to unsigned
 long integer) function [2-108](#)
- struct alignment [1-86](#)
- `strxfrm` (transform string using
 `LC_COLLATE`) function
 [2-110](#)
- `switch` [1-96](#)
- switches
 - mode selection [1-10](#)
- syntax-only (just check syntax)
 - compiler switch [1-39](#)
- system header files [1-110](#)
- system registers
 - read/write [1-84](#)
- system request values [1-106](#)
- T**
 - T (linker description file) compiler
 switch [1-39](#)
 - `tan` (tangent) function [2-112](#)

tanh (hyperbolic tangent) function [2-113](#)
 template for asm() in C programs [1-59](#)
 terminate (See atexit, exit functions)
 -threads (enable thread-safe build) compiler switch [1-39](#)
 -time (tell time) compiler switch [1-40](#)
 tokens, string convert (See strtok function)
 tolower (convert from uppercase to lowercase) function [2-114](#)
 toupper (convert characters to upper case) function [2-115](#)
 -tpautooff (no automatic templates) C++ mode compiler switch [1-46](#)
 -traditional (traditional compilation) compiler switch [1-20](#)
 transferring
 function arguments and return value [1-121](#)
 transformation functions [3-6](#)
 transformations [3-4](#)
 -trdforinit (traditional initialization) C++ mode compiler switch [1-46](#)
 Triangle window [3-72](#)
 true (See Boolean type support keywords (bool, true, false))
 twidfft2d function [3-103](#)
 twidfftrad2 function [3-99](#)

twidfftrad4 function [3-101](#)
 -typename (support typename) C++ mode compiler switch [1-46](#)

U

-U (undefine macro) compiler switch [1-25](#), [1-40](#)
 -unsigned-char (make char unsigned) compiler switch [1-40](#)
 upper case (See isupper, toupper functions)
 uppercase characters [2-58](#)
 user header files [1-110](#)

V

-v (version & verbose) compiler switch [1-40](#)
 va_arg (get next argument in variable list) function [2-116](#)
 va_arg macro [2-116](#)
 va_end (reset variable list pointer) function [2-118](#)
 va_end macro [2-118](#)
 va_start (set variable list pointer) function [2-119](#)
 va_start macro [2-119](#)
 var (variance) function [3-105](#)
 variable length arrays [1-75](#)
 variable-length argument list finishing [2-118](#)
 initializing [2-119](#)
 variance (VAR) function [3-105](#)
 vector functions [3-18](#)
 vector.h header file [3-18](#)

INDEX

VECTOR_FOR pragma [1-89](#)
-verbose (display command line)
 compiler switch [1-40](#)
-version (display version) compiler
 switch [1-40](#)
VIDL source text [1-109](#)
Visual DSP++ compiler (ccblkfn)
 guide [1-2](#)
Viterbi decoder [1-80](#)
Viterbi history and decoding
 functions [1-80](#)
volatile and asm() C program
 constructs [1-69](#)
von Hann window [3-74](#)

W

-w (disable all warnings) switch [1-42](#)
-W (override error) compiler switch
 [1-41](#)
warning messages [1-75](#)
-warn-protos (prototypes warning)
 compiler switch [1-41](#)

-wchar (support wide char type)
 C++ mode compiler switch
 [1-46](#)
-Wdriver-limit (maximum process
 errors) compiler switch [1-41](#)
-Werror-limit (maximum compiler
 errors) compiler switch [1-41](#)
white space character test (See
 isspace function)
window generators [3-18](#)
window.h header file [3-18](#)
-Wremarks (enable diagnostic
 warnings) compiler switch [1-41](#)
-write-files (enable driver I/O pipe)
 compiler switch [1-42](#)
writing
 preprocessor macros [1-110](#)
-Wterse (enable terse warnings)
 compiler switch [1-42](#)

Z

zero_cross (count zero crossing)
 function [3-106](#)